Systemprogrammierung

Skript zur Vorlesung an der RWTH-Aachen

Lehrstuhl für Informatik IV
Prof. Dr. O. Spaniol

Mesut Güneş
Ralf Wienzek
Klaus Macherey

Vorwort

Neben der Hardware bildet die Software die wichtigste Komponente eines Computers. Dabei unterscheidet man zwei Arten von Software: die *Systemsoftware*, auch *Betriebssystem* genannt, und die *Anwendungssoftware*. Die Systemsoftware umfasst alle Programme, die dem Betrieb und der Verwaltung des Rechners und der Peripherie dienen. Das Betriebssystem verwaltet somit die vorhandenen Ressourcen, wie z.B. Drucker oder Festplatten und bildet eine Schnittstelle zur Interaktion des Benutzers mit dem Rechner. Im Gegensatz hierzu wird Anwendersoftware für spezielle Anwendungen wie z.B. Tabellenkalkulation, Textverarbeitung etc. konzipiert.

Das vorliegende Skript *Systemprogrammierung* beschäftigt sich ausschließlich mit Konzepten für den Entwurf und die Realisierung von Betriebssystemen. Im Mittelpunkt der Betrachtungen stehen dabei der Begriff des Prozesses sowie Fragestellungen zur Koordination und Synchronisation nebenläufiger Prozesse.

Das Skript basiert im Wesentlichen auf der gleichnamigen Vorlesung, welche von Prof. Otto Spaniol im Wintersemester 1995/1996 gelesen wurde, ist jedoch vom Inhalt als auch in der Darstellung an die Vorlesung vom Wintersemester 2001/2002 angepasst worden.

Aachen im April 2002
Otto Spaniol
Mesut Güneş
Ralf Wienzek
Klaus Macherey

Inhaltsverzeichnis

Abbildungsverzeichnis

TEIL I

Einführung

KAPITEL 1

Was ist ein Betriebssystem?

1.1 Der Begriff des Betriebssystems

Das **Betriebssystem** eines Rechners ist ganz allgemein ein **Programm**, das sich zwischen dem Nutzer und der Hardware einordnen lässt. Ein **Betriebssystem** soll dem Benutzer eine Umgebung bereitstellen, in der er Programme ausführen kann, und zwar in komfortabler und effizienter Weise. Um diese Aufgabe zu erfüllen, muss ein Betriebssystem die ihm zur Verfügung stehenden Hardware-**Ressourcen** wie z.B. **CPU-Zeit**, **Speicher** und Kommunikationskanäle verwalten und dem Anwender unter Beachtung gewisser Kriterien wie Effizienz oder Fairness zur Verfügung stellen (siehe Abbildung 1.1).

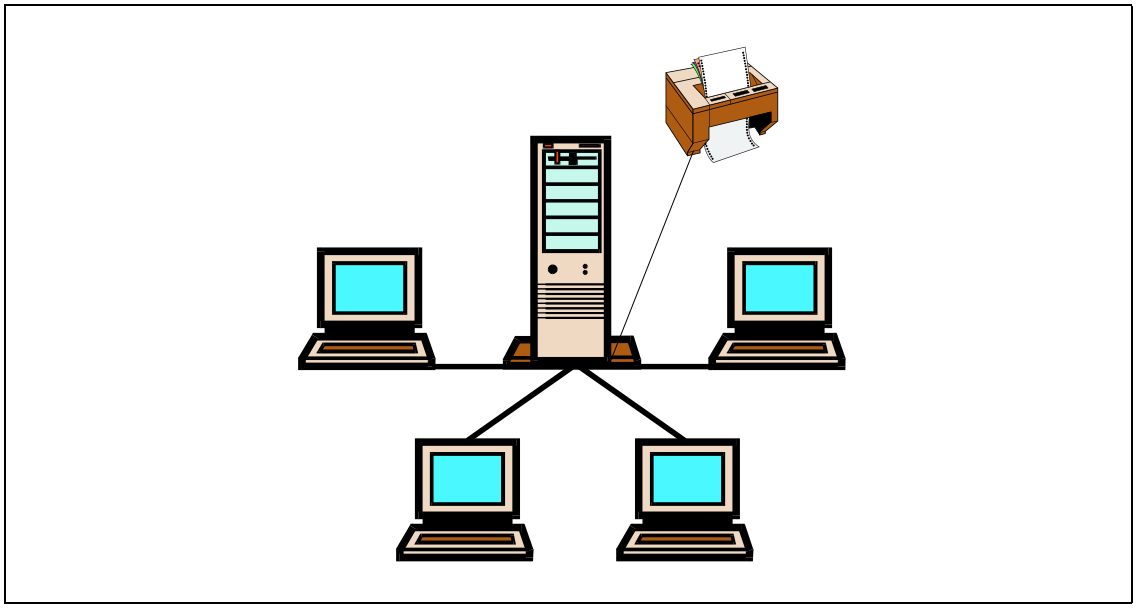


Abbildung 1.1: Umfeld des Betriebssystems

1.2 Zur Geschichte der Betriebssysteme

Die ersten Computer kannten noch kein Betriebssystem. Sämtliche Aufgaben, die heutzutage ein Betriebssystem übernimmt, mussten vom Programmierer manuell durchgeführt

werden. Beispielsweise erfolgte das Laden eines Programms durch direktes Eingeben binärer Befehle in den Speicher oder schon komfortabler über **Lochkarten**. Später wurden diese Aufgaben durch zusätzliche Hard- und Software erleichtert. Es wurden Lochkartenleser zum Laden und Speichern und **Assembler** zur Programmierung eingeführt. Trotzdem war die Zeitdauer, die allein für die Vorbereitung eines Programmlaufs notwendig war, noch sehr groß. Dies hatte zur Folge, dass die CPU der damaligen Rechner nur ineffizient genutzt werden konnte. Mit dem Aufkommen der ersten höheren Programmiersprachen wie **FORTRAN** und **COBOL** wurde es möglich, Programme, die z.B. den gleichen **Compiler** benötigten, direkt hintereinander ablaufen zu lassen. Dadurch konnte ein erneutes Laden des Compilers vermieden werden. Dieser so genannte **Batch-Betrieb** wurde zuerst manuell und später automatisch über einen residenten **Monitor** durchgeführt. Trotz dieser Verbesserungen blieb die **CPU-Auslastung** immer noch gering. Der Grund lag in der Geschwindigkeitsdiskrepanz zwischen der CPU und den oft mechanischen Ein-/Ausgabe-Geräten (**E/A-Geräten**). Dieser Unterschied konnte jedoch durch eine Entkopplung der CPU von den langsamen E/A-Geräten verringert werden ⇒ **Offline-Betrieb** (siehe Abbildung 1.2). Zu diesem Zweck wurden modernere **Magnetbänder** benutzt, um z.B. Ausgaben der CPU an den Drucker zwischenzuspeichern. Ein späterer Wechsel des Bandes ermöglichte so den Ausdruck der Daten bei gleichzeitiger Weiterarbeit der CPU mit einem neuen Magnetband.

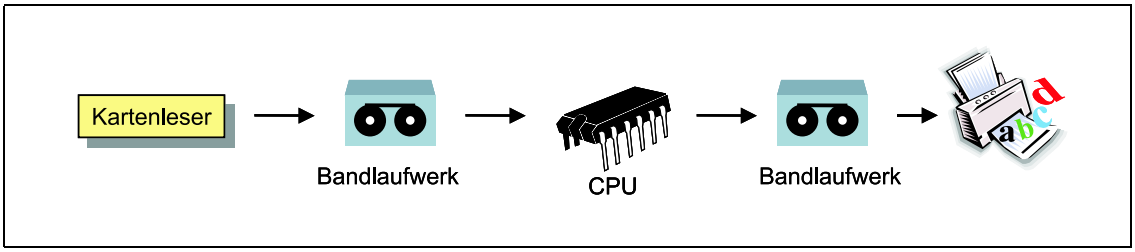


Abbildung 1.2: Operationsformen mit E/A-Geräten Offline

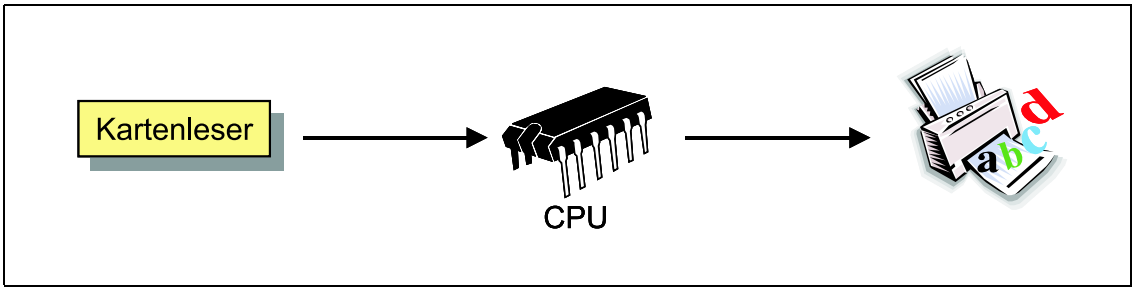


Abbildung 1.3: Operationsformen mit E/A-Geräten Online

Mit der Entwicklung von Plattenspeichern entstand eine neue Art der Abarbeitung von Jobs: das so genannte **Spooling**[1]. Die Platte diente dabei als **Puffer** für Ein- und Ausgabeoperationen. So war es möglich, dass die CPU weiterarbeitete, während z.B. Daten von der Platte gedruckt oder von Kartenlesern auf Platte gespeichert wurden (siehe Abbildung 1.3). Spooling führte schließlich dazu, mehrere Jobs *gleichzeitig* ausführen zu können, der so genannte **Multiprogramming**-Betrieb, in dem Sinn, dass der aktuell auf der CPU arbeitende Job diese freigab, sobald er eine E/A-Operation machte, mit der eine längere Wartezeit verbunden war. Die CPU konnte dann in der Zwischenzeit von einem anderen Job benutzt werden, wodurch sich Leerlaufzeiten der CPU nahezu vollständig

[1] Simultaneous Peripheral Operation On-Line

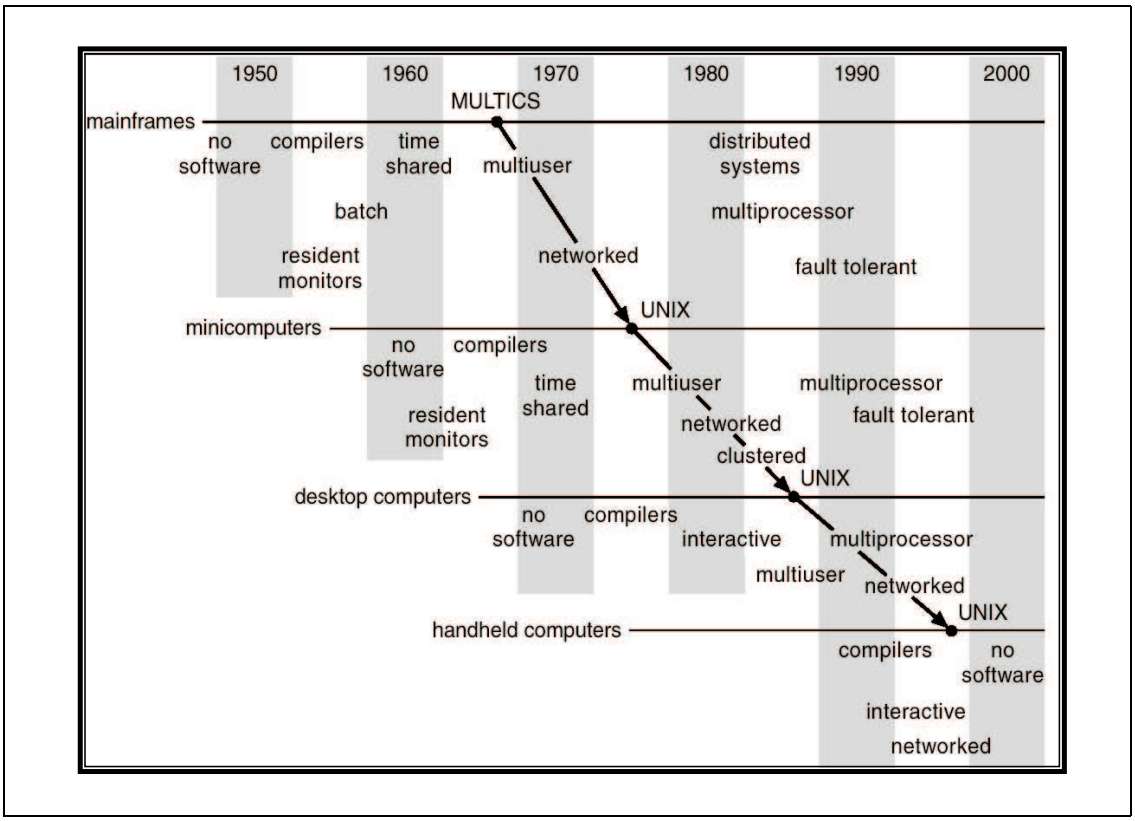


Abbildung 1.4: Entwicklung der Betriebssysteme (Aus Silberschatz, 6. Auflage)

vermeiden ließen. Mit dem **Time-Sharing** oder **Multitasking** wurde eine Variante des **Multiprogramming** entwickelt, bei dem ein Wechsel der Jobs jeweils nach Ablauf einer gewissen Zeitspanne, auch **Zeitscheibe** genannt, erfolgt. Dadurch ließ sich z.B. die CPU-Zeit gerechter unter den Jobs aufteilen, da eine Monopolisierung des Systems durch rechenintensive Jobs vermieden werden konnte. Time-Sharing-Betriebssysteme eigneten sich insbesondere für den Multiuser-Betrieb eines Rechners: Mehrere Benutzer konnten damit *gleichzeitig* eine CPU interaktiv für ihre Arbeit nutzen. In den letzten Jahren zeichnen sich neben der Verbesserung der Einprozessor-Betriebssysteme zwei weitere Trends ab: Zum einen stellt die Entwicklung von Multiprozessorsystemen, d.h. Parallelrechnern, neue Anforderungen an Betriebssysteme. Zum anderen ermöglicht die zunehmende Vernetzung der Rechner die Nutzung verteilter Ressourcen, für die verteilte Betriebssysteme notwendig werden.

Ein Beispiel für die Fortschritte im Bereich der Betriebssysteme ist das Betriebssystem **MULTICS**, welches am Massachusetts Institute of Technology (MIT) für den dortigen Großrechner entwickelt wurde. Später gebaute Mini- und Microcomputer nutzten die Ideen von MULTICS, und so entstand das weit verbreitete **UNIX**-Betriebssystem (Bell Labs). Abbildung 1.4 stellt die Entwicklung der Betriebssysteme über die Jahrzehnte graphisch dar.

1.3 Aufbau eines Rechners

Die Struktur heutiger Computersysteme ist in Abbildung 1.5 dargestellt. Die CPU ist über den Systembus mit den Kontrolleinheiten (Device-Controller) verschiedener Geräte (z.B.

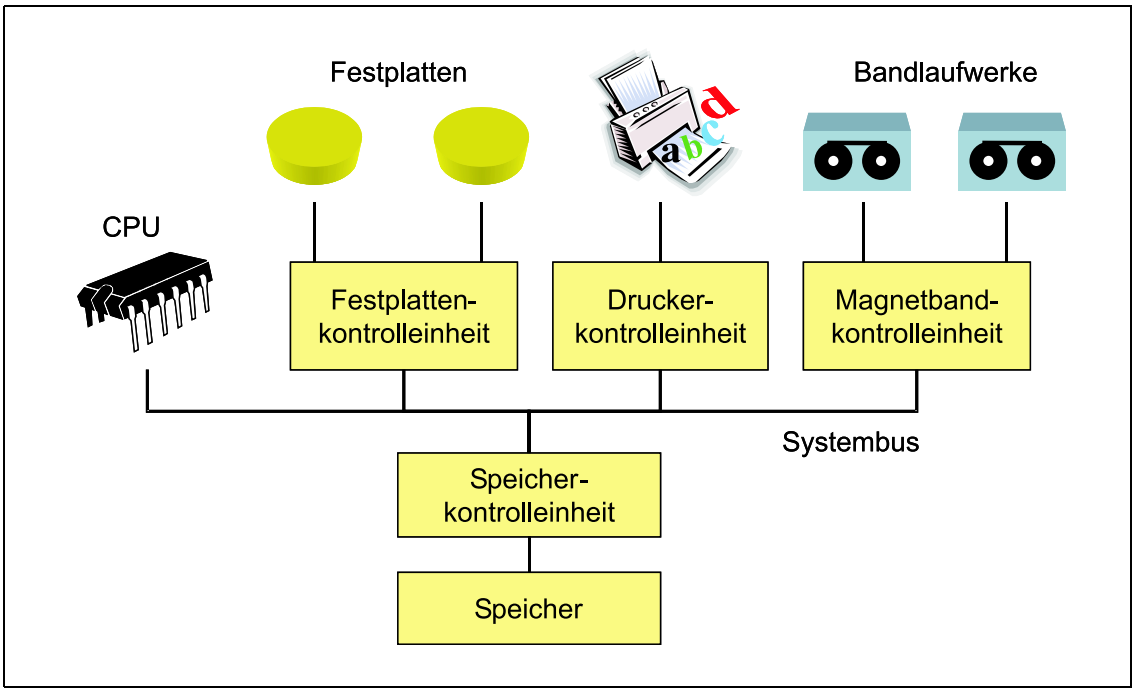


Abbildung 1.5: Struktur moderner Computersysteme

Platte, Drucker oder Speicher) verbunden. Jede Kontrolleinheit steuert den Zugriff auf das entsprechende Gerät. Dies ist z.B. dann notwendig, wenn mehrere Stellen gleichzeitig auf ein Gerät zugreifen. Außerdem besitzen die Kontrolleinheiten Puffer, mit denen die Geschwindigkeitsunterschiede zwischen den verschiedenen Teilen des Rechners ausgeglichen werden können. Nach dem Einschalten eines Computers wird zuerst ein Boot-Programm

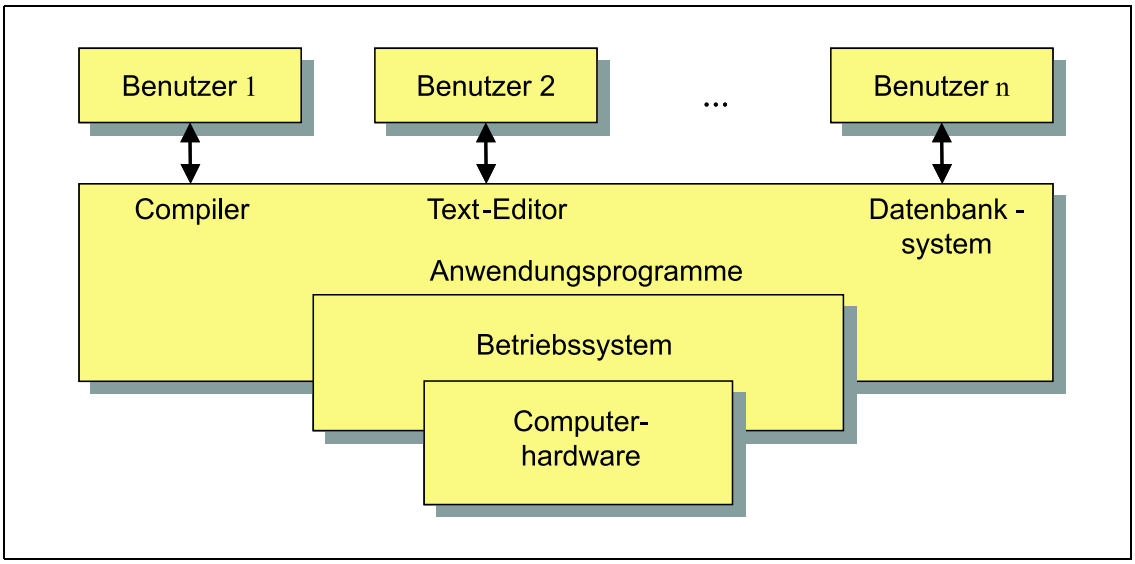


Abbildung 1.6: Aufbau eines Computersystems aus der Softwareperspektive

gestartet, um das System (CPU-Register, Controller, Speicher ...) zu initialisieren. Danach wird das Betriebssystem in den Speicher geladen und gestartet. Anschließend wartet das System, bis ein *Ereignis* eintritt. Ereignisse werden der CPU mit Hilfe von Interrupts signalisiert. Initiator eines Interrupts kann entweder die Hardware (Drucker: „Ich bin bereit!“) oder die Software (Programm: „Schreibe auf Festplatte!“) sein. Im Falle ei-

nes Interrupts stoppt die CPU ihre aktuelle Arbeit, speichert ihren Zustand und startet eine dem Interrupt entsprechende Routine. Nach der Ausführung dieser Routine wird die vorher unterbrochene Arbeit an der gleichen Stelle wieder aufgenommen. Der Aufbau eines Computersystems aus der Softwareperspektive ist in Abbildung 1.6 dargestellt.

Grundsätzlicher Ablauf einer E/A-Operation

Das Beispiel in Abbildung 1.7 soll den Ablauf der Interruptbehandlung am Beispiel einer einfachen E/A-Operation verdeutlichen. Ein Benutzerprozess fordert über einen System-

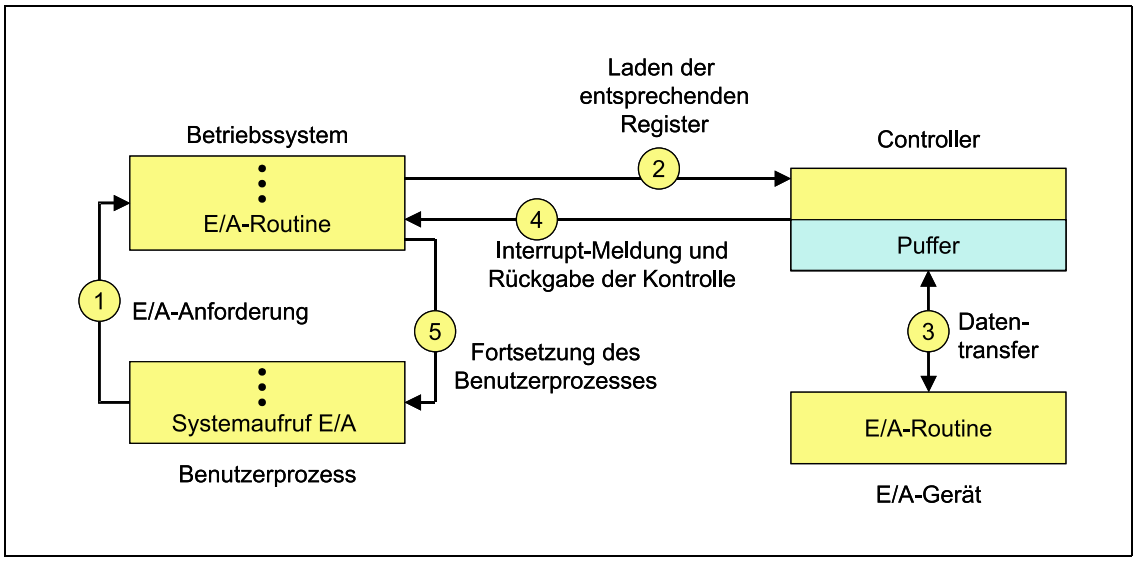


Abbildung 1.7: Grundsätzlicher Ablauf einer E/A-Operation

aufruf (**Interrupt**) eine E/A-Operation an ①. Der Systemaufruf unterbricht den Benutzerprozess und gibt die Kontrolle an das Betriebssystem, welches die entsprechende E/A-Routine aufruft. Diese Routine lädt die notwendigen Werte in die Register und den Puffer des Controllers und stößt die E/A-Operation des Controllers an ②. Die CPU kann sich während des Datentransfers ③ anderen Aufgaben widmen oder auf dessen Beendigung warten. Durch einen Interrupt ④ meldet das E/A-Gerät das Ende der E/A-Operation. Das Betriebssystem ruft eine entsprechende Interrupt-Routine auf und setzt den zuvor unterbrochenen Prozess fort ⑤.

Synchrone und asynchrone Ein-/Ausgabe

Wird nach der Initiierung des Datentransfers ③ die Kontrolle an das Betriebssystem zurückgegeben, so spricht man von **asynchroner E/A**. Asynchrone E/A führt zu einer Effizienzsteigerung, da die CPU trotz E/A weiterarbeiten kann. Allerdings kommt es auch zu erhöhtem Verwaltungsaufwand, da das Betriebssystem Kenntnis über die Zustände mehrerer E/A-Geräte und Prozesse haben muss. **Synchrone E/A** wartet nach dem Anstoßen einer E/A-Operation, bis diese beendet wird. Erst dann wird die Kontrolle an das Betriebssystem zurückgegeben. Der Hauptvorteil synchroner E/A liegt in ihrer Einfachheit.

Direct Memory Access (DMA)

Da sehr oft große Datenmengen z.B. zwischen den Puffern der E/A-Geräte und dem Hauptspeicher bewegt werden müssen, ist die CPU häufig ausschließlich mit Datentransport-Befehlen beschäftigt. Um sie von dieser Arbeit zu entlasten, kann **Direct Memory Access** (DMA) verwendet werden. DMA ermöglicht es einer E/A-Kontrolleinheit, die Übertragung der Daten auf eigene Faust, d.h. nahezu unabhängig von der CPU, durchzuführen. Die CPU initialisiert dazu zunächst die entsprechenden Register des E/A-Controllers. Der eigentliche Datentransport wird vom Controller eigenständig bewerkstelligt. Die so gewonnene Zeit kann von der CPU zur Erledigung anderer Aufgaben verwendet werden.

Speicherstrukturen

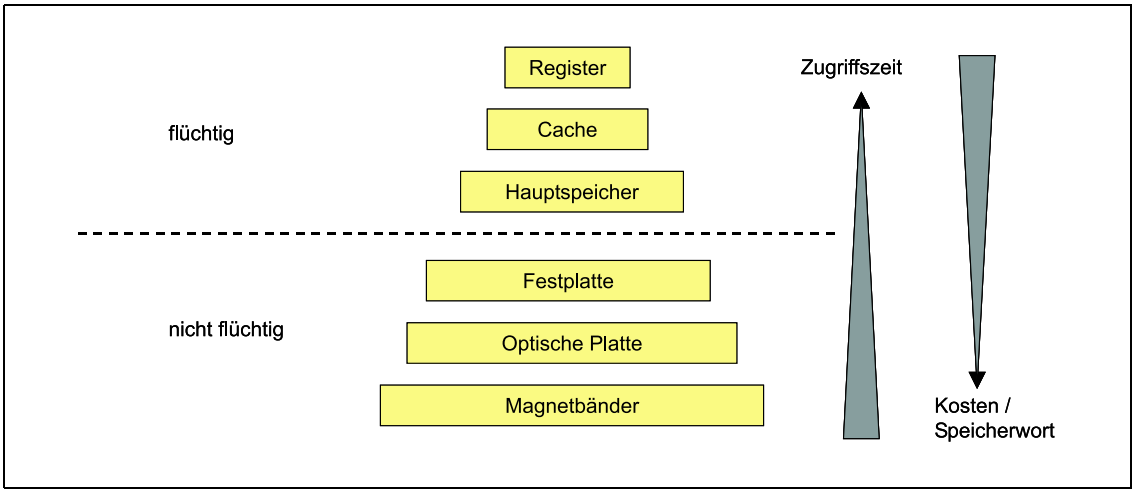


Abbildung 1.8: Typische Speicherhierarchie

Damit ein Computer ein Programm ausführen kann, muss sich das Programm in einem **Speicher** befinden, auf den die CPU direkt zugreifen kann. Außerdem muss der Speicher so groß sein, dass das entsprechende (Teil-) Programm in ihm Platz hat. Beide Bedingungen werden vom Hauptspeicher eines Computers erfüllt. Trotz dieser Eigenschaften hat der Hauptspeicher auch Nachteile.

- Er ist sehr viel langsamer als die CPU,
- er ist flüchtig, d.h. die Daten sind nur vorhanden, solange der Rechner eingeschaltet ist, und
- er ist i.A. nicht groß genug, um zusätzliche Daten zu speichern, die momentan nicht benötigt werden.

Diese Probleme werden durch einen hierarchischen Aufbau des Speichers gelöst (siehe Abbildung 1.8). An der Spitze dieser Hierarchie stehen sehr schnelle Speicher, die häufig benötigte Daten speichern (**Register**, **Cache**). Da diese Art des Speichers pro Speicherwort relativ teuer ist, haben Register und Cache nur eine geringe Kapazität.

Um Daten dauerhaft zu speichern, werden permanente Speichermedien (**Festplatte**, **Optische Platte**) als Sekundär- oder **Hintergrundspeicher** benutzt. Die Zugriffszeit auf den Sekundärspeicher ist zwar wesentlich größer als die Zugriffszeit auf den Hauptspeicher, dafür kann dort jedoch ein Vielfaches an Daten gespeichert werden.

1.4 Aufbau und Komponenten eines Betriebssystems

Wie uns der kurze Blick in die Geschichte zeigte, sind Betriebssysteme heutzutage hochkomplexe Programme, die eine Vielzahl von Aufgaben erfüllen müssen. Um die Realisierung zu vereinfachen, kann ein Betriebssystem in einzelne Komponenten (siehe Abbildung 1.9) unterteilt werden, die man (relativ) unabhängig voneinander entwickeln kann.

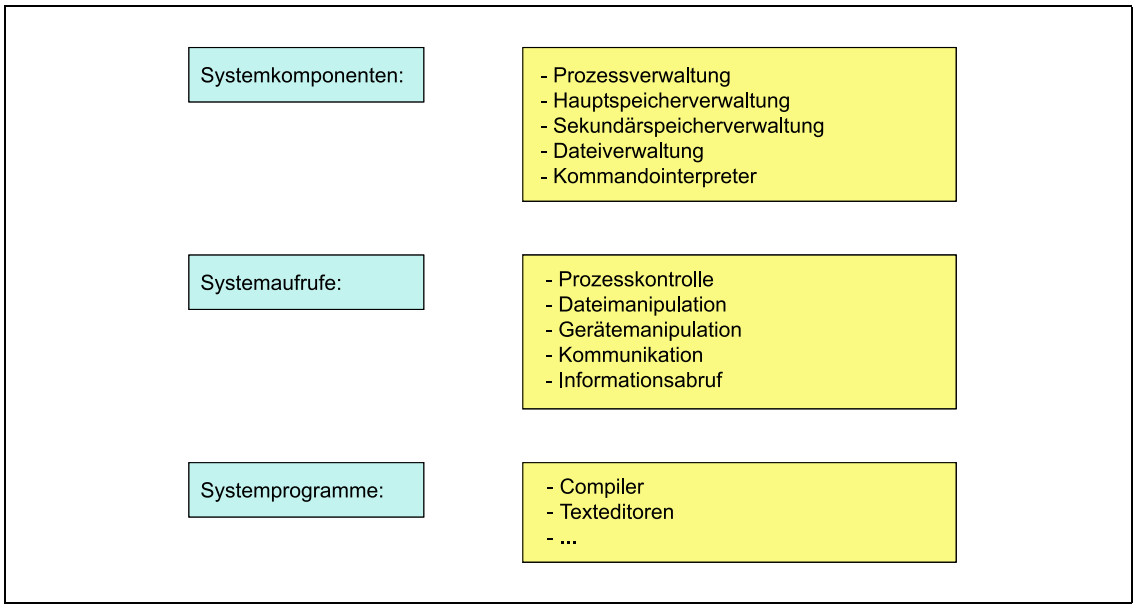


Abbildung 1.9: Komponenten des Betriebssystems

Prozessverwaltung

Ein **Prozess** ist vereinfacht gesagt ein **Programm** in Ausführung. Sind auf einem Rechner mehrere Prozesse *gleichzeitig* aktiv, so müssen diese vom Betriebssystem verwaltet werden. Ein Betriebssystem muss

- neue Prozesse erzeugen und alte vernichten,
- die zur Verfügung stehende CPU-Zeit auf die Prozesse verteilen (Scheduling),
- Prozesse, die z.B. gemeinsame Variablen benutzen, synchronisieren, um Inkonsistenzen zu vermeiden,
- Verklemmungen (Deadlocks), d.h. kein Prozess kann weiterarbeiten, beheben und
- Mittel zur Interprozess-Kommunikation zur Verfügung stellen.

Hauptspeicherverwaltung

Ein Prozess benötigt für seine Arbeit Speicherplatz. Vor seiner Ausführung müssen daher alle relevanten Daten in den **Hauptspeicher** geladen werden. Aufgabe des Betriebssystems ist es,

- die belegten und freien Speicherbereiche zu verwalten,

- zu entscheiden, welcher neue Prozess bei freiwerdendem Speicherplatz geladen wird, und
- zu entscheiden, wieviel Speicherplatz einem Prozess zugewiesen wird (Zuteilung/-Entzug).

Sekundärspeicherverwaltung

Eigentlich ist diese der Hauptspeicherwaltung recht ähnlich. Ein Unterschied zwischen Haupt- und Sekundärspeicher liegt jedoch in der Zugriffszeit. Diese dauert beim **Sekundärspeicher** nicht nur länger, sondern kann insbesondere stark variieren[2]. Dadurch ergeben sich für die Sekundärspeicherverwaltung zusätzlich Aufgaben wie:

- Disk Scheduling (mehrere Anfragen können beispielsweise gesammelt werden und dann in einer günstigen Reihenfolge abgearbeitet werden),
- Speicherplatzzuteilung, da die Lage der Daten direkten Einfluss auf die Zugriffszeit hat, sowie
- Verwaltung der freien Speicherbereiche.

Dateiverwaltung

Die Dateiverwaltung abstrahiert von der physikalischen Speicherung der Daten. Dazu werden diese logisch als Dateien und Verzeichnisse angesehen, die dem Anwender die Arbeit mit den Daten vereinfachen. Das Betriebssystem ermöglicht dabei

- das Erstellen, Manipulieren und Löschen von Dateien bzw. Verzeichnissen,
- eine geeignete Abbildung der logischen auf die physikalische Struktur des Speichers und
- ein automatisches Sichern wichtiger Daten als Backup.

Kommandointerpreter

Ein **Kommandointerpreter** bildet eine Schnittstelle zwischen Betriebssystem und Benutzer. Sie bietet dem Benutzer eine Möglichkeit, die Funktionalität des Betriebssystems zu nutzen, z.B. Speicherung einer Datei, Start eines Programms usw. Kommandozeilen-Interpreter lesen dazu die Texteingabe eines Benutzers und starten eine entsprechende Aktion. Andere Interpreter stellen dafür Icons oder Menüs zur Verfügung. Der Kommandointerpreter ist entweder als Teil des Betriebssystems oder als separates Systemprogramm realisiert.

Systemaufrufe

Über Systemaufrufe können Prozesse direkt mit dem Betriebssystem kommunizieren. Der Aufruf erfolgt dabei über eine Bibliotheksprozedur. Diese hat einen Namen und je nach

[2] Bewegung des Lese/Schreibkopfes bei einer Platte.

Typ eine Reihe von Parametern. Realisiert sind Bibliotheksprozeduren meist in Assembler. Es gibt jedoch auch Systeme, die dafür höhere Programmiersprachen (wie z.B. C) benutzen.

BEISPIEL

Systemaufruf

Benötigt ein Prozess während der Ausführung eines Programms Daten aus einer Datei, so muss er die Datei öffnen und die entsprechenden Daten lesen. Eine solche Leseoperation könnte in einem C-Programm wie folgt aussehen:

```
count = read(file, buffer, n-bytes);
```

Der Systemaufruf hat den Namen `read`, übergibt die drei Parameter `file`, `buffer` und `n-bytes` und gibt als Resultat des Aufrufs einen Wert zurück, der in einer Variable `count` gespeichert wird.

Bedeutung des Aufrufs: Lies eine Anzahl von Bytes (`n-bytes`) aus der Datei (`file`) in einen Puffer (`buffer`). Nach dem Aufruf gibt der Wert `count` an, wie viele Bytes tatsächlich gelesen wurden. Normalerweise gilt `count = n-bytes`, beim Erreichen des Dateiendes (`EndOfFile`) oder bei fehlerhaftem Aufruf sind jedoch auch andere Werte denkbar.

Systemaufrufe können grob in fünf Kategorien eingeteilt werden. Typische Beispiele für Aufrufe der jeweiligen Bereiche sind:

1. Prozesskontrolle:

- create/terminate process
- wait/signal event
- allocate/free memory

2. Dateimanipulation:

- create/delete file
- open/close file
- read/write file

3. Gerätemanipulation:

- request/release device
- read/write/reposition device

4. Kommunikation:

- create/delete communication connection
- send/receive message

5. Abruf von Informationen:

- get process, get file, get data attributes
- get/set time, get/set date

Systemprogramme

Systemprogramme sollen dem Benutzer die Arbeit mit dem Computer erleichtern. Sie sollen ihm helfen, Programme zu erstellen, sie auszuführen usw. Um diese Unterstützung zu erreichen, stellen Betriebssysteme eine Reihe von Programmen zur Verfügung, die direkt auf das Betriebssystem aufsetzen und dessen Komplexität vor dem Benutzer verbergen. Sie bieten eine im Vergleich zu den Systemaufrufen komfortable Schnittstelle. Zu den Programmen, die häufig als Systemprogramme realisiert sind, gehören Kommandointerpreter, z.B. `shell` bei UNIX, `command.com` bei MS-DOS, Compiler und einfache Texteditoren. Systemprogramme sind nicht Teil des Betriebssystems. Beim Kauf eines Betriebssystems werden jedoch häufig die wichtigsten Systemprogramme mitgeliefert.

1.5 Das Schichtenkonzept

In diesem Abschnitt wird eine oft verwendete Vorgehensweise vorgestellt, welche bei der Realisierung von aufwendigen Systemen eingesetzt wird. Anwendung findet diese Vorgehensweise, z.B. beim Entwurf von Betriebssystemen und in der Datenkommunikation. Dabei wird ein komplexes System in **Schichten** aufgeteilt (siehe Abbildung 1.10). Die unteren Schichten sind hierbei eher *hardwarenah* und die oberen *anwendungsnah*. Eine Schicht nutzt jeweils die Funktionalität, die ihr von einer darunterliegenden Schicht geboten wird, und bietet ihrerseits der darüberliegenden Schicht Dienste an.

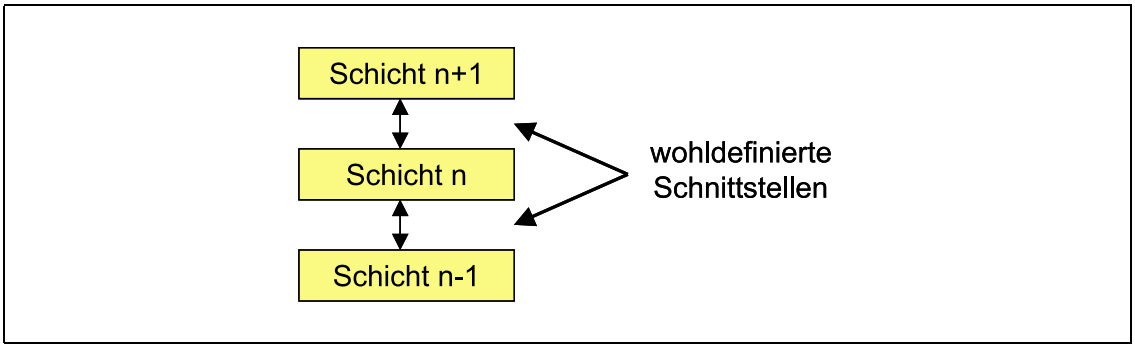


Abbildung 1.10: Prinzipielle Struktur einer Schichtenarchitektur

Der Vorteil des Schichtenkonzeptes ist, dass bei fester Schnittstellendefinition die Realisierung einer Schicht problemlos ausgetauscht werden kann. Der Nachteil des Schichtenprinzips liegt in seiner aufwendigen Realisierung.

Schichtenkonzepte in der Datenkommunikation

Die Kommunikation zwischen Rechnern erfolgt über so genannte Protokolle. Diese legen fest, wie Daten, die über ein Netzwerk von einem anderen Rechner ankommen, zu interpretieren sind. Da die hierfür benötigte Software sehr komplex ist, wird auch hier ein Schichtenkonzept verwendet (siehe Abbildung 1.11):

Eine Anwendung eines Rechners A, die mit einer Anwendung des Rechners B kommunizieren möchte, gibt die Daten an ihre oberste Kommunikationsschicht weiter. Diese Schicht, die durch ein Protokoll mit der entsprechenden Schicht des Rechners B kommuniziert, leitet die Daten an die darunterliegenden Schichten weiter. Die Daten werden schließlich auf

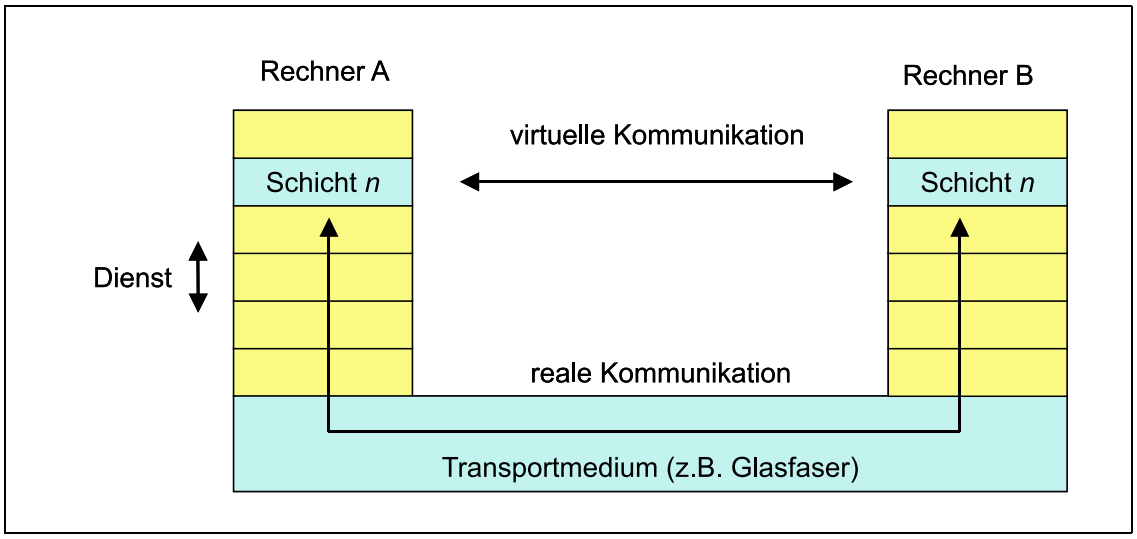


Abbildung 1.11: Nutzung einer Schichtenarchitektur bei der Rechnerkommunikation

das Netz gegeben und durchlaufen beim Rechner B die Schichten in umgekehrter Reihenfolge. Es erfolgt also eine virtuelle Kommunikation zwischen gleichen Schichten, die reale Kommunikation findet jedoch mit der darüber- bzw. darunterliegenden Schicht statt.

Das ISO/OSI-Referenzmodell

Das ISO/OSI-Referenzmodell[3] basiert auf einem Vorschlag, der von der ISO für die Standardisierung von Kommunikationsprotokollen entwickelt wurde. Es besteht aus 7 Schichten und dem Kommunikationsmedium. Die Bezeichnung der einzelnen Schichten sowie ein Versuch, die verschiedenen Funktionalitäten anhand einer *Eselsbrücke* zu veranschaulichen, findet sich in Abbildung 1.12.

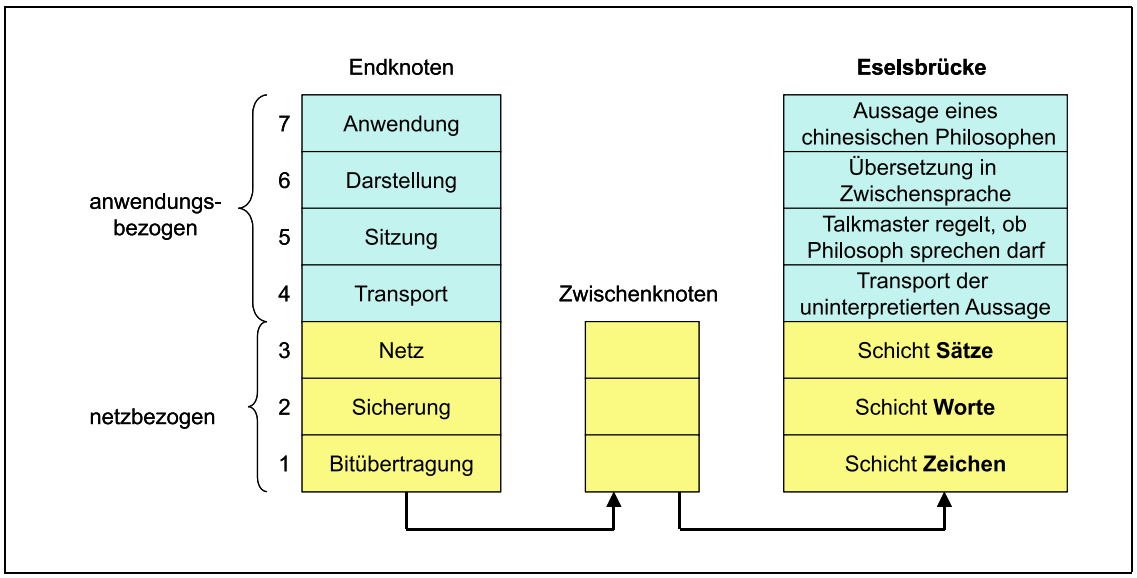


Abbildung 1.12: Kommunikationsstandardisierung gemäß ISO/OSI

[3]International Standard Organization/Open Systems Interconnection

1.6 Beispielarchitekturen

Nachdem bisher die wesentlichen Konzepte und Komponenten von Betriebssystemen beschrieben wurden, wollen wir nun einige real existierende Betriebssysteme vorstellen.

Windows

Windows aus dem Hause Microsoft ist mit seinen unterschiedlichen Versionen (95/ME-/NT/2000/XP) das am weitesten verbreitete Betriebssystem für Personal-Computer. Während seiner Evolution von DOS bis WindowsXP erlebte Windows sehr unterschiedliche Erweiterungen und Entwicklungen. Die Struktur von Windows 2000 ist in Abbildung 1.13 dargestellt. Die Struktur ist nach dem Schichtenprinzip aufgebaut und modular. Die

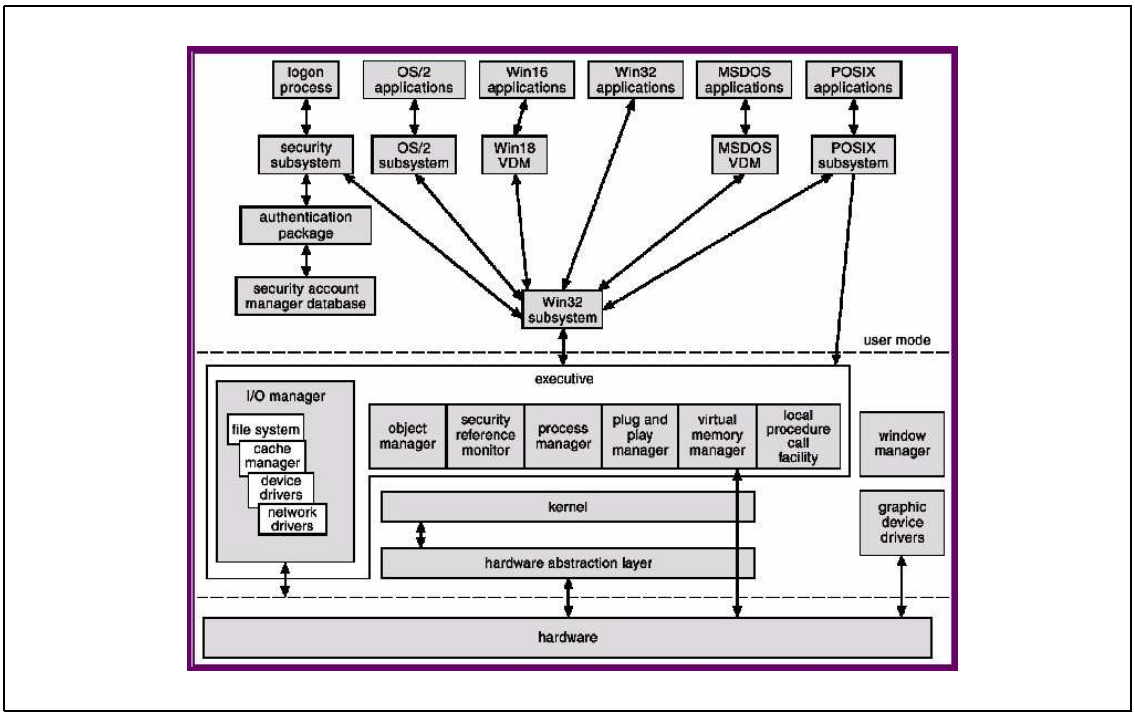


Abbildung 1.13: Die Systemstruktur von Windows 2000

wichtigsten Schichten sind das **Hardware-Abstraction Layer**, der **Kernel** und die **Executive-Schicht**. Alle diese Schichten werden im so genannten **Protected-Mode** ausgeführt, wodurch ihnen das gesamte Befehlsspektrum zur Verfügung steht. Zusätzlich zu diesen *lebenswichtigen* Komponenten existiert eine Vielzahl von Subsystemen, die im User-Mode ausgeführt werden und eingeschränkten Zugriff auf die Ressourcen des Systems haben.

UNIX/Linux

UNIX/Linux ist ein **Multiuser-Multitasking** Betriebssystem, welches seit Mitte der 70er Jahre entwickelt wird und sehr verbreitet ist. Es gibt UNIX-Versionen für die unterschiedlichsten Rechnertypen. Die Systemstruktur von UNIX ist in Abbildung 1.14 dargestellt und besteht aus drei Schichten dem **Kernel**, den **System-Libraries** und den **System-Utilities**. Der eigentliche Betriebssystemkern von UNIX, der Kernel, wird im

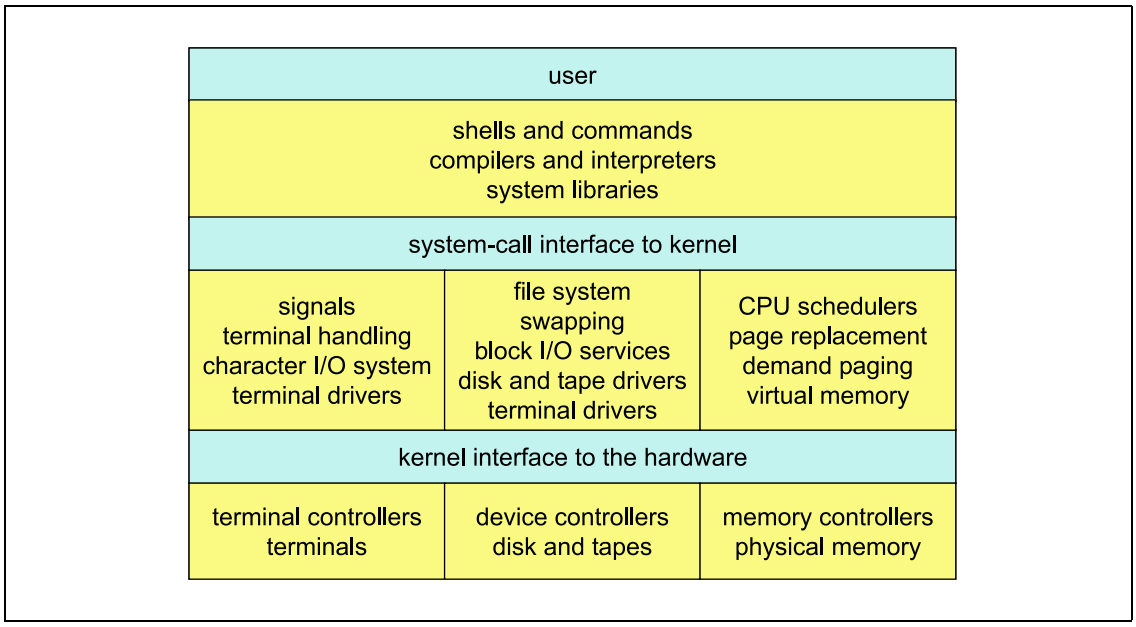


Abbildung 1.14: Die Systemstruktur von UNIX

Supervisor-Modus ausgeführt. Dadurch wird das System vor allzu eifrigen Benutzern oder Prozessen geschützt. Die System-Libraries bieten eine klar definierte Schnittstelle für Anwendungen, die im User-Modus laufen und nur eingeschränkten Zugriff auf die System-Ressourcen haben.

TEIL II

Prozessverwaltung

KAPITEL 2

Prozesse

Der Begriff des Prozesses stellt eines der grundlegendsten Konzepte im Bereich der Systemprogrammierung dar. In der Einleitung wurde ein Prozess kurz als „Programm im Stadium der Ausführung“ definiert, um dadurch den dynamischen Charakter eines Prozesses im Gegensatz zur „Statik“ eines Programmcodes zum Ausdruck zu bringen.

Der Prozessbegriff lässt sich sowohl auf die Benutzer als auch auf die Systemebene anwenden. In den meisten modernen Rechensystemen gibt es eine größere Anzahl von nebeneinander existierenden Benutzer- und Systemprozessen. Jeder Prozess benötigt für seine Ausführung bestimmte Ressourcen, wie etwa Speicherplatz, CPU-Zeit und Zugriff auf E/A-Geräte. Greifen nun Prozesse simultan auf das gleiche Betriebsmittel zu, kann es durch den konkurrierenden Zugriff zu Konflikten kommen. Die effiziente Koordination derartiger Aktivitäten ist eine Grundaufgabe der Systemprogrammierung.

Prozesse befinden sich in unterschiedlichen Zuständen, die sich durch die jeweilige Art der Aktivität charakterisieren lassen. Solche Zustände sind beispielsweise:

Zustand	Bedeutung
running	Anweisungen des Prozesses werden gerade ausgeführt
ready	Prozess ist bereit zur Ausführung, wartet aber noch auf freien Prozessor
waiting	Prozess wartet auf das Eintreten eines Ereignisses, z.B. darauf, dass ein von ihm belegtes Betriebsmittel fertig wird
blocked	Prozess wartet auf ein fremdbelegtes Betriebsmittel
new	Prozess wird erzeugt (kreiert)
killed	Prozess wird (vorzeitig) abgebrochen
terminated	Prozess ist beendet, d.h. alle Instruktionen sind abgearbeitet

Wichtig ist in diesem Zusammenhang, dass zu jedem Zeitpunkt auf einem beliebigen Prozessor immer nur ein Prozess laufen kann, wohingegen sich viele Prozesse gleichzeitig im Ready- oder Waiting-Status befinden können.

Die Übergänge zwischen den einzelnen Zuständen kann man sich leicht anhand eines Diagramms veranschaulichen. Ein Beispiel für ein solches **Prozesszustandsdiagramm** zeigt Abbildung 2.1 (ohne die Zustände blocked und killed):

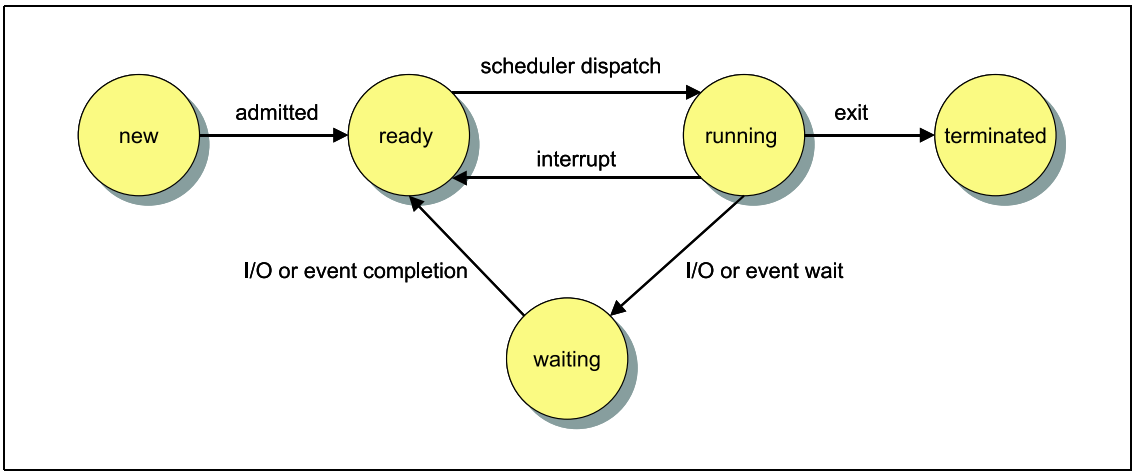


Abbildung 2.1: Prozesszustände und Zustandsübergänge

Im Zusammenhang mit Prozessen ist das Betriebssystem vor allem zuständig für das Erzeugen und Beenden von Benutzer- und Systemprozessen, die Aufteilung von Speicherplatz und CPU-Zeit, die Bereitstellung von Mechanismen zur Synchronisation und Kommunikation unter Prozessen sowie zum Vorgehen in Deadlock-Situationen.

Dazu müssen Prozesse eine Repräsentation innerhalb des Betriebssystems besitzen. Diese erfolgt über so genannte **Prozesskontrollblöcke** (PCB), die alle für das Betriebssystem relevanten Informationen über einen Prozess beinhalten. In einem PCB finden sich beispielsweise folgende Informationen:

PCB-Feld	Bedeutung
Status	Prozesszustand
Programmzähler	enthält die Adresse der nächsten auszuführenden Instruktion
CPU-Scheduling	hier finden sich z.B. Prioritäten, Zeiger auf Warteschlangen und ähnliche Scheduling-Parameter
Speichermanagement	enthält die letzten Werte der Speicherregister, Seitentabellen und ähnliche Informationen für die Speicherverwaltung
E/A-Status	Liste von zugeordneten E/A-Geräten, offenen Files usw.
Accounting	benötigte CPU- und Realzeit, Zeitbeschränkungen, Prozessnummern u.ä.

2.1 Scheduling

Betriebssysteme lassen sich aufgrund ihrer Arbeitsweise grob in Singletasking-, Multiprogramming- und Multitasking-Systeme einteilen. Beim **Singletasking** enthält der Hauptspeicher den Betriebssystemkern und darauf aufsetzend den **Kommandointerpreter** (Command Interpreter, CI). Ein Prozess wird bei seinem Aufruf exklusiv in den Speicher eingelesen. Bei der Implementierung ist zu beachten, dass der Kommandointerpreter beim Einlesen des Prozesses u.U. überschrieben werden kann.

Multiprogramming und **Multitasking** unterscheiden sich davon insofern, dass jetzt neben dem Betriebssystemkern und dem Kommandointerpreter mehrere Prozesse gleichzeitig im Speicher koexistieren können (siehe Abbildung 2.2):

Das Betriebssystem verwaltet also mehrere Prozesse gleichzeitig und regelt insbesonde-

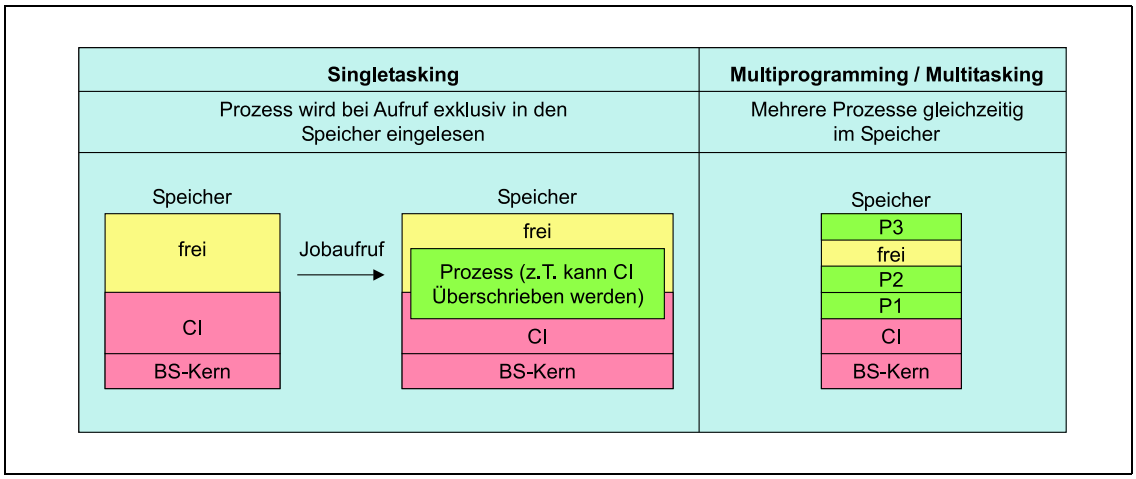


Abbildung 2.2: Prinzip des Single- und Multitasking

re den Zugriff auf die CPU. Dieses so genannte **Scheduling** ist entscheidend für einen sorgsamen Umgang mit der verfügbaren CPU-Zeit. Scheduling findet auf verschiedenen zeitlichen Ebenen statt: Der Langzeit- oder **Job-Scheduler** ist dabei zuständig für die Auswahl der Prozesse, die in den Speicher geladen werden, während der Kurzzeit- oder **CPU-Scheduler** entscheidet, welcher der (bereits geladenen) Prozesse im Ready-Status wann auf die CPU zugreifen darf. Gute Scheduling-Strategien legen Wert auf eine ausgewogene Mischung (Job-Mix) zwischen rechenintensiven und E/A-intensiven Prozessen, um die Geschwindigkeitsdiskrepanz zwischen den langsameren E/A-Geräten und der wesentlich schnelleren CPU auszugleichen.

Der Unterschied zwischen Multiprogramming und Multitasking besteht darin, dass beim Multiprogramming der Benutzer während seiner CPU-Nutzung nicht unterbrochen werden kann (was natürlich bei rechenintensiven Prozessen zu einer Blockade der CPU führen kann), während beim Multitasking (auch **Time-Sharing** genannt) ein Prozess sowohl durch einen Interrupt als auch nach Ablauf einer bestimmten Zeitspanne unterbrochen werden kann. Dies bietet mehreren Benutzern die Möglichkeit zu gleichzeitigem interaktiven Betrieb, wobei jeder einzelne Benutzer die Illusion hat, die CPU arbeite nur für ihn. Ein Beispiel hierfür ist das Round-Robin-Verfahren, auf das wir in Kapitel 6 genauer eingehen werden.

2.2 Interprozesskommunikation

Sobald man die Koexistenz mehrerer Prozesse erlaubt, müssen auch Möglichkeiten zur Kommunikation zwischen diesen Prozessen vorgesehen werden. Hierzu gibt es zwei unterschiedliche Ansätze. Beim **Message-Passing** (siehe Abbildung 2.3) wird zunächst über das Betriebssystem eine Verbindung zwischen Prozess P_1 und Prozess P_2 aufgebaut. Typische Systemaufrufe in diesem Zusammenhang sind etwa `gethostid`, `open-connection`, `accept-connection` und `close-connection`. Die so eingerichtete Verbindung kann zu zwei oder mehreren Prozessen gehören, unterschiedliche Kapazität aufweisen, unterschiedliche Nachrichtenlängen zulassen, uni- oder bidirektional sein und auf verschiedenartigste Weise implementiert werden. Mit Hilfe von `send`- und `receive`-Befehlen werden dann über diese Verbindung Nachrichten ausgetauscht.

Der andere Ansatz **Shared-Memory** (siehe Abbildung 2.4) beruht darauf, die exklusive

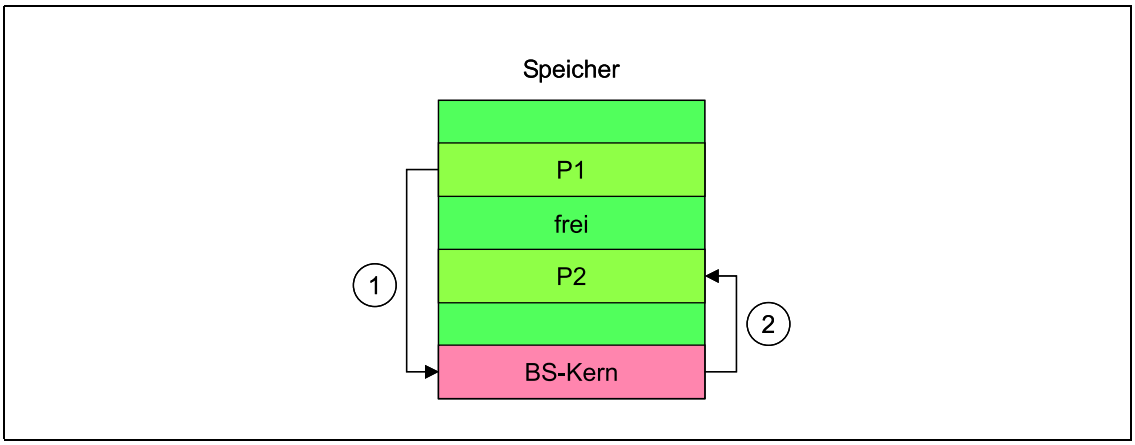


Abbildung 2.3: Prinzip des Message-Passing

Nutzung von Speicherbereichen durch einzelne Prozesse zeitweise aufzuheben. Prozess P_1 speichert dabei eine Nachricht an Prozess P_2 ohne Umweg über den Betriebssystemkern in einem Speicherbereich, auf den auch P_2 Zugriff hat.

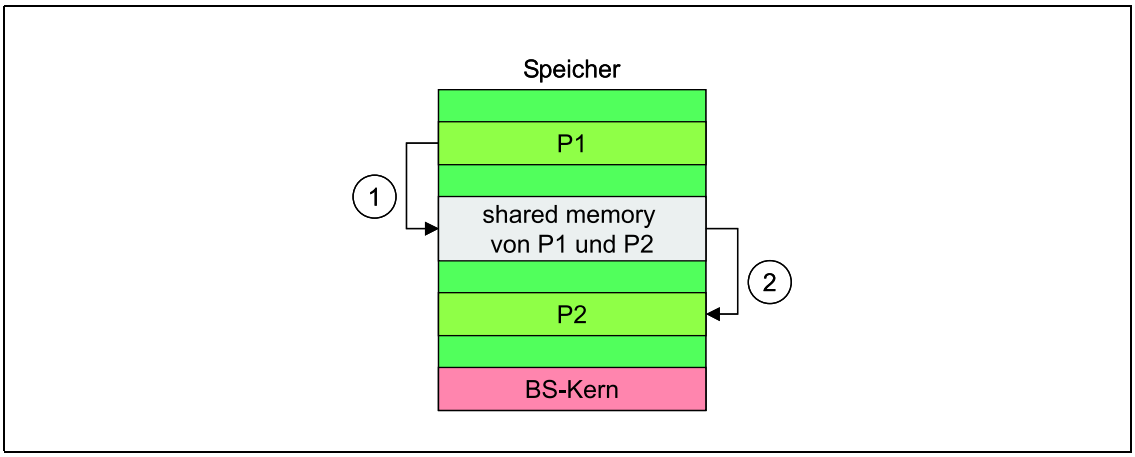


Abbildung 2.4: Prinzip des Shared-Memory

Diese Vorgehensweise ist vor allem bei großen Datenmengen schneller als das Message-Passing. Allerdings können sehr leicht Synchronisations- bzw. Konsistenzprobleme auftreten, wenn etwa P_2 bereits liest, während P_1 noch schreibt. Mechanismen zur Umgehung solcher Konflikte werden wir im Kapitel 3 über das wechselseitige Ausschlussproblem ausführlich kennenlernen.

2.3 Threads

Bislang haben wir den Begriff des Prozesses betrachtet. Dabei haben wir einen Prozess als ein Programm im Stadium der Ausführung charakterisiert. Ein Prozess wird durch die ihm zugeordneten Ressourcen und den Ort der Ausführung definiert. Im vorhergehenden Abschnitt haben wir Möglichkeiten zur Kommunikation zwischen Prozessen kennengelernt. Ein Wechsel zwischen Prozessen in einem Multitasking-Betriebssystem kann mitunter sehr teuer sein, da ein vollständiger Wechsel der Prozessumgebung stattfinden muss. Es gibt daher viele Anwendungungen, in denen das Sharing und der (konkurrierende) Zu-

griff auf Ressourcen *innerhalb* eines Prozesses (also innerhalb des gleichen Adressraums) wünschenswert ist. Dieses Konzept bezeichnet man als **Thread** (Faden).

2.3.1 Thread-Varianten

Mit der Einführung des Thread-Begriffs kommen wir zu folgender Begriffsdefinition:

- Eine Einheit, die den Eigentümer von Ressourcen bezeichnet, wird (weiterhin) Prozess genannt.
- Eine Einheit, die eine gewisse Aufgabe erledigt, wird Thread oder leichtgewichtiger Prozess (Lightweight-Process, LWP) genannt.

Bislang entsprach ein Prozess genau einem Thread. Im Folgenden wollen wir nun Multithread-Umgebungen betrachten, d.h. innerhalb eines Prozesses können jetzt mehrere Threads ausgeführt werden. Jeder Thread besteht dabei aus folgenden Komponenten:

- Einem Zustand der Threadausführung, z.B. Running, Ready, ...
- Thread-Kontext, dieser muss gesichert werden, falls sich der Thread nicht im Zustand Running befindet
- Stack für die Ausführung
- Speicherplatz für lokale Variablen
- Zugriff auf Speicher und Ressourcen des zugehörigen Prozesses, die mit allen anderen Threads des gleichen Prozesses geteilt werden.

Insbesondere werden also die Code-Segmente und Daten-Segmente sowie die Ressourcen zwischen Threads des gleichen Prozesses geteilt.

2.3.2 Singlethreading

Zur Laufzeit werden die Prozessregister vom Prozess kontrolliert. Befindet sich der Prozess nicht im Zustand „Running“, so wird der Inhalt der Register gespeichert.

2.3.3 Multithreading

Die Zuordnung von PCB und User Address Space zum Prozess bleibt bestehen, aber jeder Thread bekommt zusätzlich einen Kontrollblock, den so genannten Thread-Control-Block (TCB), der für jeden Thread die Register, den Zustand und andere Thread-bezogene Informationen enthält.

Dieses Konzept führt dazu, dass alle Threads eines Prozesses den gleichen Adressraum nutzen und auf die gleichen Daten zugreifen. Legt z.B. ein Thread das Ergebnis einer Rechnung im Adressraum ab, so können alle anderen Threads dieses Prozesses ebenfalls darauf zugreifen. Hierdurch können natürlich auch Sicherheitsprobleme entstehen.

Ein Vorteil des Thread-Konzeptes ist, dass zur Generierung eines Threads in einem existierenden Prozess wesentlich weniger Zeit benötigt wird, da die für Prozesse typischen

zeitaufwendigen Kontextwechsel entfallen. Es müssen nur die Thread-bezogenen Informationen gesichert werden. Ein weiterer Vorteil besteht in der vereinfachten und effizienteren Kommunikation zwischen Threads eines Prozesses untereinander, da hierfür der Betriebssystemkern nicht mehr involviert wird.

Neben den Vorteilen von Threads gibt es auch einige Nachteile wie z.B. die bereits angedeutete Sicherheitsproblematik:

- Durch den gemeinsam genutzten Adressraum sind die Daten einzelner Threads nicht vor dem Zugriff anderer Threads des gleichen Prozesses geschützt.
- Erfolgt ein Swapping des übergeordneten Prozesses, so werden auch alle Threads des Prozesses mit ausgelagert. Terminiert der übergeordnete Prozess, so terminieren auch alle dem Prozess zugeordneten Threads.

2.3.4 Threadzustände

Analog zu Prozessen sind die grundlegenden Zustände eines Threads „Running“, „Ready“ und „Blocked“. Zustände zur Suspendierung von Threads werden nicht benötigt, da aufgrund des gemeinsam genutzten Adressraums aller Threads die Suspendierung einzelner Threads nicht sinnvoll ist.

2.3.5 User-Level Threads

Bei **User-Level Threads** ist das Thread-Management Aufgabe der Anwendung. Der Betriebssystemkern braucht also von der Existenz solcher Threads nichts zu wissen. Jede Anwendung kann als multithreaded programmiert werden, indem so genannte Thread-Bibliotheken genutzt werden. Diese Thread-Bibliotheken sind eine Sammlung von Routinen, die gerade zum Zwecke des User-Level Thread-Management bereitgestellt werden.

Vorteile

- Die Thread-Wechsel + Thread-Erzeugung benötigen keine Privilegien des Systemmodus, da das Thread-Management innerhalb des Nutzeradressraumes eines einzelnen Prozesses stattfindet. Damit wird Zeit für Modiwechsel (System-/Nutzermodus) eingespart.
- Das Scheduling kann anwendungsspezifisch realisiert werden.
- Dieser

  Mechanismus kann in jedem Betriebssystem ablaufen, ohne dass Betriebssystemkern-Erweiterungen vorgenommen werden müssen.

Nachteile

- Wird ein Thread blockiert, so blockiert gleichzeitig der gesamte Prozess, d.h. auch die Menge aller anderen Threads.
- Reine User-Level Threads können in einer Multiprozessor-Umgebung nicht parallel ausgeführt werden, da der Betriebssystemkern einem Prozess auch nur einen Prozessor zuordnet.

2.3.6 Kernel-Level Threads

In einer reinen **Kernel-Level** Thread-Umgebung wird das Thread-Management vom Betriebssystemkern durchgeführt. Dabei kann jede Anwendung als multithreaded programmiert werden.

Vorteile

- Deutlich schneller in Erzeugung / Wechsel als Prozess.
- Der Betriebssystemkern kann mehrere Threads eines Prozesses auf verschiedenen Prozessoren einer Multiprozessorumgebung ausführen.
- Falls ein Thread blockiert, kann die Kontrolle einem anderen Thread desselben Prozesses übergeben werden.

Nachteile

- Wird die Kontrolle von einem Thread eines Prozesses an einen anderen Thread desselben Prozesses übergeben, so ist jedesmal ein Moduswechsel erforderlich, d.h. die Ausführung führt zu einer Verlangsamung.

KAPITEL 3

Koordination und Synchronisation nebenläufiger Prozesse

In aller Regel arbeiten Prozesse heute nicht mehr isoliert, sondern kooperieren mit anderen Prozessen. Ihre Ausführung erfolgt also quasi-gleichzeitig, z.B. ineinander verzahnt (interleaved). Bei der Kooperation mehrerer Prozesse kann es leicht dazu kommen, dass ein Prozess P_1 auf einen anderen Prozess P_0 angewiesen ist. Beispielsweise erzeugt P_0 Resultate, deren Ausgabe auf einem Endgerät durch P_1 veranlasst wird. Abstrakter ausgedrückt heißt dies: Prozess P_0 erzeugt etwas, das Prozess P_1 verbraucht. Das Erzeuger-Verbraucher-Problem wird uns als erstes klassisches Beispiel für Fragen der Koordination und Synchronisation nebenläufiger Prozesse dienen. Eine schematische Beschreibung des Problems ist in Abbildung 3.1 dargestellt.

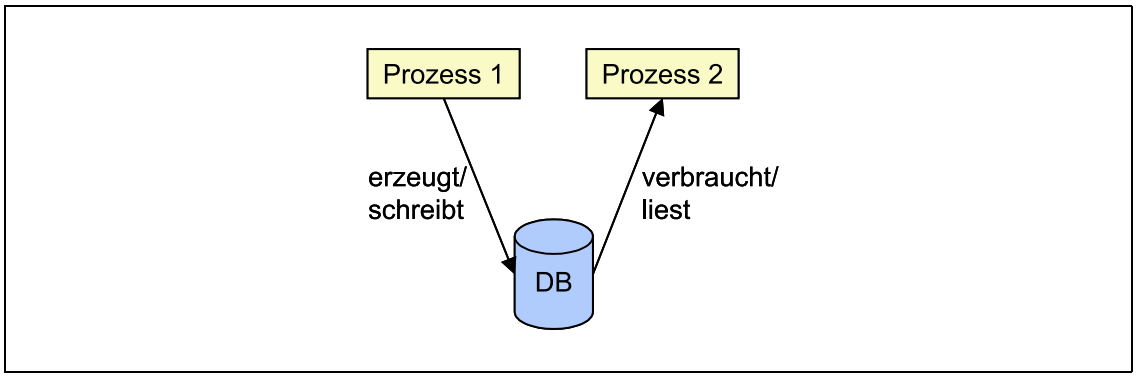


Abbildung 3.1: Schematische Darstellung des Erzeuger-Verbraucher-Problems

3.1 Das Erzeuger-Verbraucher-Problem

Gegeben seien die Prozesse P_E (Erzeuger) und P_V (Verbraucher). Ferner sei zwischen P_E und P_V ein Lager mit endlicher Kapazität MAX (etwa in Form eines Pufferspeichers) eingerichtet. Der Erzeuger kann Produkte im Zwischenlager ablegen und der Verbraucher kann Produkte daraus entnehmen. Dabei setzen wir voraus, dass die produzierten bzw. konsumierten Einheiten jeweils genau einen Pufferplatz beanspruchen.

Für das Zusammenspiel von Erzeuger und Verbraucher gelten folgende Randbedingungen:

1. Wenn das Lager voll ist, kann P_E nichts ablegen.
2. Wenn das Lager leer ist, kann P_V nichts entnehmen.
3. Der Lagerbestand kann nicht gleichzeitig von P_E und P_V verändert werden.

Bedingung 3, die allgemeine Konsistenzbedingung des Problems, kann bei einfachem Drauflosprogrammieren sehr leicht unter den Tisch fallen. Dies passiert beispielsweise, wenn wir das Erzeuger-Verbraucher-Problem durch Verwendung zweier Routinen `ERZEUGER` und `VERBRAUCHER` angehen, die beide auf eine gemeinsame Variable `S`, den aktuellen Lagerbestand, zugreifen und folgende Gestalt haben:

PROGRAMM

Erzeuger

```
repeat
  erzeuge Element;
  while S = MAX do
    noop;
  lege Element in Puffer ab;
  S := S + 1;
until FALSE;
```

PROGRAMM

Verbraucher

```
repeat
  while S = 0 do
    noop;
  entnimm Element aus Puffer;
  S := S - 1;
  verbrauche Element;
until FALSE;
```

Beide Routinen laufen unabhängig voneinander. Daher kann es bei unkontrolliertem Zugriff auf den Lagerbestand `S` zu Inkonsistenzen kommen, nämlich dann, wenn P_E nach dem Überprüfen der Bedingung `S = MAX` ein Produkt ablegt, aber nicht mehr dazu kommt, den Lagerbestand zu korrigieren, weil in diesem Moment P_V auf den Lagerbestand `S` zugreift, eine Einheit entnimmt, das alte `S` um 1 verringert und erst dann wieder P_E das Feld überlässt (siehe Abbildung 3.2). P_E führt dann seine angefangene Routine zu Ende und erhöht seinerseits das alte (und inzwischen leider veraltete) `S` um 1.

Obwohl sich also am Lagerbestand de facto nichts geändert hat, ist das neue `S` größer als das alte. Die Korrektur des Lagerbestands durch P_V wurde nicht berücksichtigt, was zu einem so genannten **Lost-Update** führt.

3.1.1 Eine einfache Lösung mittels Ringpuffer

Eine Lösung des Erzeuger-Verbraucher-Problems, die diese Schwierigkeiten umgeht, verwendet einen **Ringpuffer**. Dieser wird mittels einer Modulo-Funktion *mod* realisiert und kann in der Realität z.B. in Form eines ringförmigen Diamagazins angetroffen werden. Zunächst sei die Lösung auf einen Erzeuger- und einen Verbraucherprozess beschränkt; später werden wir auch den Fall mehrerer Erzeuger und Verbraucher einbeziehen.

Die Idee ist einfach: Der Erzeuger legt sein Produkt im ersten freien Pufferplatz ab, der

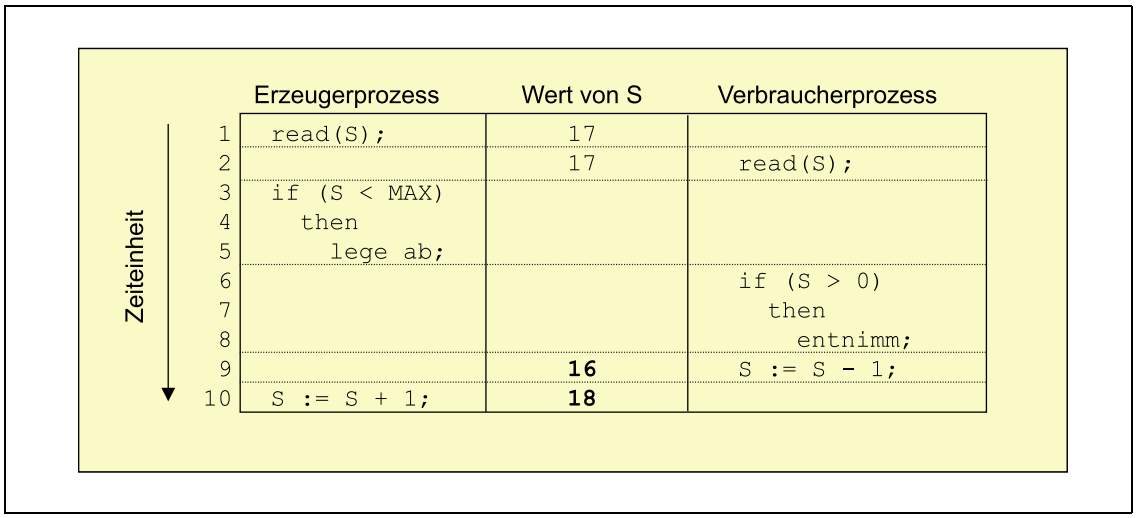


Abbildung 3.2: Einfaches Beispiel für das Auftreten von Inkonsistenzen

Verbraucher entnimmt ein Produkt aus dem ältesten belegten. Beide vereinbaren, dass der Puffer höchstens MAX-1 belegte Komponenten haben soll.

Zur Realisierung werden zwei Variablen benötigt (vgl. Abbildung 3.3): `in` zeigt auf den ersten freien Puffererplatz (mod MAX) und `out` zeigt auf den ersten belegten Pufferplatz (mod MAX).

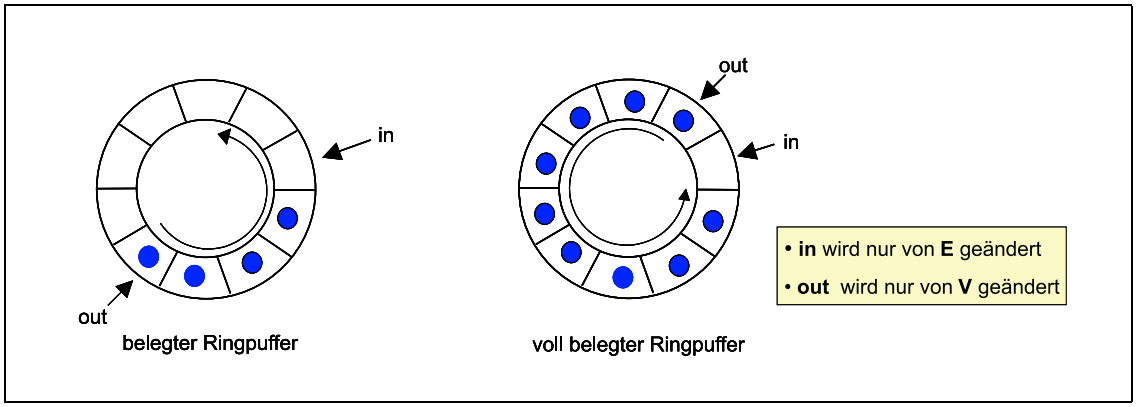


Abbildung 3.3: Lösung des Erzeuger-Verbraucher-Problems mittels Ringpuffer

Der Puffer ist mithin genau dann leer, wenn `in = out` gilt, und genau dann voll, wenn `in+1 mod MAX = out` ist (man beachte, dass höchstens MAX-1 Pufferplätze belegt werden dürfen). Erzeuger und Verbraucher lassen sich wie folgt implementieren:

PROGRAMM

Erzeuger

```
repeat
  produziere ein Element und lege es in nextp ab; (* Zwischenlager   *)
  while (in+1 mod MAX = out) do                    (* Puffer voll?    *)
    noop;
  buffer[in] := nextp;                             (* ablegen         *)
  in := in+1 mod MAX;                              (* Zeigerkorrektur *)
until FALSE;
```

PROGRAMM

Verbraucher

```
repeat
  while (in = out) do                              (* Puffer leer?    *)
    noop;
  nextc := buffer[out];                            (* entnehmen       *)
  out := out + 1 mod MAX;                          (* Zeigerkorrektur *)
  verbrauche das in nextc enthaltene Element;
until FALSE;
```

3.1.2 Ein allgemeiner Lösungsansatz und neue Schwierigkeiten

Kann man diese Lösung in einfacher Weise auf den Fall mehrerer Erzeuger- und mehrerer Verbraucherprozesse verallgemeinern, indem man den Ringpuffer als allen gemeinsames Lager der Kapazität MAX ansieht und den Zugriff darauf über eine globale Zählvariable `counter` realisiert? Setzt man den Zähler zu Beginn auf 0 (bzw. einen beliebigen Anfangsfüllstand zwischen 0 und MAX), so lauten die vorgeschlagenen Algorithmen:

PROGRAMM

Erzeuger

```
repeat
  produziere ein Element und lege es in nextp ab;  (* Zwischenlager    *)
  while (counter = MAX) do                         (* Lager voll?     *)
    noop;
  buffer[in] := nextp;                             (* ablegen         *)
  in := in + 1 mod MAX;                            (* Zeigerkorrektur *)
  counter := counter + 1;                          (* Zaehlerkorrektur *)
until FALSE;
```

PROGRAMM

Verbraucher

```
repeat
  while (counter = 0) do                           (* Lager leer?     *)
    noop;
  nextc := buffer[out];                            (* entnehmen       *)
  out := out + 1 mod MAX;                          (* Zeigerkorrektur *)
  counter := counter - 1;                          (* Zaehlerkorrektur *)
  verbrauche das in nextc enthaltene Element;
until FALSE;
```

Diese Lösung ist prinzipiell in Ordnung, krankt allerdings (wie auch die Lösung des einfachen Falles) daran, dass Probleme mit dem Füllstand auftreten können. Der Grund dafür liegt im Befehl `counter := counter +/- 1`. Eine maschinensprachliche Implementierung dieses Befehls hat üblicherweise die folgende Form:

PROGRAMM

Implementierung von counter := counter +/- 1

```
register1 := counter;
register1 := register1 +/- 1;
counter := register1;
```

Dabei ist `register1` ein lokales CPU-Register. Der eine programmiersprachliche Befehl

besteht also aus mehreren maschinensprachlichen Anweisungen. Inkrementiert nun ein Prozess den Zähler, während ihn der andere gleichzeitig dekrementiert, stellt sich dies auf der Maschinenebene als sequenzielle Verzahnung der entsprechenden Anweisungen dar, die unter widrigen Umständen z.B. folgendes Aussehen hat (zu Beginn sei `counter = 5`):

BEISPIEL

Verzahnung von Anweisungen

Taktzyklus	Prozess	Anweisung	Resultat
1	E	`register1 := counter`	register1 = 5
2	E	`register1 := register1 + 1`	register1 = 6
3	V	`register2 := counter`	register2 = 5
4	V	`register2 := register2 - 1`	register2 = 4
5	E	`counter := register1`	counter = 6
6	V	`counter := register2`	counter = 4

Es wurde also je eine Einheit erzeugt und verbraucht, dennoch ändert sich der Füllstand. Der Grund für solche Konsistenzprobleme ist stets darin zu suchen, dass gemeinsame Datenbestände von mehreren Prozessen gleichzeitig manipuliert werden. Ziel der folgenden Überlegungen wird sein, solche unkontrollierten simultanen Manipulationen zu verhindern.

3.2 Das Problem des wechselseitigen Ausschlusses

Gegeben seien n Prozesse $P_0, P_1, ..., P_{n-1}$. Bei jedem dieser Prozesse lässt sich der Programmcode in kritische und unkritische Bereiche aufteilen. Ein **kritischer Bereich** ist dabei definiert als Phase, in der ein Prozess gemeinsame (globale) Daten benutzt. Die anderen Phasen des Prozesses nennt man analog dazu **unkritische Bereiche** (vgl. Abbildung 3.4).

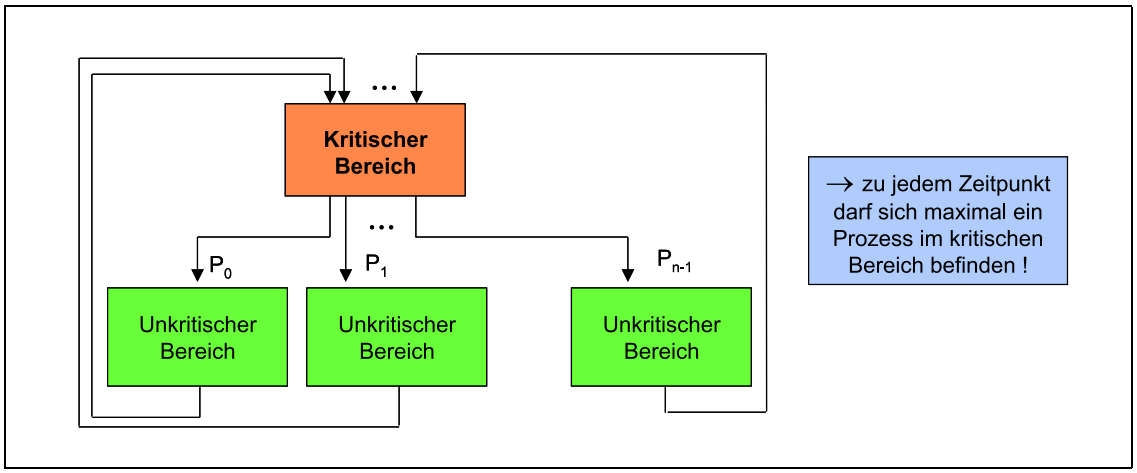


Abbildung 3.4: Kritische und unkritische Bereiche

Befinden sich nun zwei Prozesse gleichzeitig in ihren jeweiligen kritischen Bereichen, so ist das Ergebnis ihrer Operationen (wie eben exemplarisch dargestellt) möglicherweise nicht mehr determiniert, sondern von zufälligen Umständen (etwa der Scheduling-Strategie) abhängig. Verhindern lässt sich dies dadurch, dass man jeweils nur einem einzigen Prozess den Zugang zu seinem kritischen Bereich gestattet. Umgekehrt heißt dies, dass kein anderer Prozess mehr seinen kritischen Bereich betreten darf, sobald ein Prozess sich in

seinem kritischen Bereich aufhält. Dies ist eine hinreichende Bedingung dafür, dass die beschriebenen Inkonsistenzen ausgeschlossen werden.

Das wechselseitige Ausschlussproblem besteht nun in der Erstellung von Algorithmen, die dies gewährleisten. Dazu stehen zwei Mechanismen zur Verfügung: eine **Sperre**, die es für jeden Prozess vor dem Betreten des kritischen Bereichs zu überwinden gilt, und ein **Freigabemechanismus**, der das Verlassen des kritischen Bereichs signalisiert. Ganz so einfach ist die Lösung jedoch nicht, denn sehr schnell stellt sich heraus, dass die Sperre und die Freigabe selbst wieder kritische Bereiche sind. Damit die Sperre funktioniert, muss sie **unteilbare (atomare) Operationen** enthalten. Dies können z.B. einfache Assemblerbefehle (wie etwa einfache Lese-/Schreibzugriffe oder Test-and-Set-Anweisungen) sein. Aus solchen einfachen atomaren Operationen lassen sich dann komplexe atomare Bereiche (Makros) konstruieren. Dies ist allein schon deshalb zweckmäßig, da die Programmierung ohne Makros (also einzig unter Verwendung von elementaren atomaren Operationen) sehr schnell unübersichtlich werden kann.

Eine korrekte Lösung des wechselseitigen Ausschlussproblems muss folgende drei Bedingungen erfüllen:

1. **Mutual-Exclusion:** Zu jedem Zeitpunkt darf sich höchstens ein Prozess in seinem kritischen Bereich aufhalten.

2. **Progress:** Befindet sich kein Prozess in seinem kritischen Bereich, und gibt es andererseits Prozesse, die ihren kritischen Bereich betreten möchten, so hängt die Entscheidung, welcher Prozess als nächster seinen kritischen Bereich betreten darf, nur von diesen Kandidaten ab und fällt in endlicher Zeit. Die Entscheidung ist also unabhängig von der Ausführungsgeschwindigkeit der Prozesse (insbesondere hat es keinen Einfluss, wenn ein Prozess in seinem unkritischen Bereich stirbt).

3. **Bounded-Waiting:** Nachdem ein Prozess sein Interesse am Betreten des kritischen Bereichs kundgetan hat, kann es vorkommen, dass zwischendurch noch andere Prozesse die Erlaubnis zum Betreten ihres kritischen Bereichs erhalten, bevor besagter Prozess an die Reihe kommt. Es muss aber möglich sein, eine endliche obere Schranke für die Zeit anzugeben, die unser Prozess warten muss, bis er dran ist.

Es sei ausdrücklich darauf hingewiesen, dass die dritte Bedingung den Ausschluss von Prioritätsregelungen beinhaltet und außerdem impliziert, dass jeder Prozess seinen kritischen Bereich in endlicher Zeit wieder verlässt (also nicht etwa darin stirbt oder in einer Endlosschleife landet). Setzt man letzteres voraus, so lässt sich Bounded-Waiting am leichtesten durch Angabe einer Schranke für die Anzahl der Prozesse nachweisen, die unserem Prozess vorgezogen werden dürfen.

3.2.1 Zwei untaugliche Versuche und eine Lösung

Die folgenden Überlegungen beschränken sich zunächst auf Algorithmen, die für zwei gleichzeitig arbeitende Prozesse P_i und P_j angewendet werden können. Eine naheliegende Idee besteht darin, einfache Lese- und Schreibzugriffe als unteilbare Operationen zu realisieren. Beispielsweise können Befehle wie `turn := j` oder `choosing[i] := FALSE` atomar sein. Damit ließe sich z.B. ein Schalter `turn` heranziehen, der anzeigt, welcher der beiden Prozesse mit dem Betreten seines kritischen Bereichs an der Reihe ist. Nachdem ein Prozess seinen kritischen Bereich verlassen hat, zeigt der Schalter auf den anderen der beiden Prozesse.

Dieser Ansatz führt zu folgendem Algorithmus für Prozess P_i (analog P_j):

PROGRAMM

Algorithmus für Prozess P_i

```
repeat
  Unkritischer Bereich;
  while (turn <> i) do
    noop;
  Kritischer Bereich;
  turn := j;
until FALSE;
```

Diese Lösung ist allerdings nicht korrekt. Falls nämlich `turn = j` gilt und P_j in seinem unkritischen Bereich stirbt, kann P_i nie mehr in seinen kritischen Bereich eintreten.

Dieses Problem lässt sich dadurch vermeiden, dass die Sperre erst dann ausgelöst wird, wenn wirklich ein Prozess in seinen kritischen Bereich möchte. Dies kann mittels einer Boole'schen Variablen `flag[i]` in jedem Prozess P_i realisiert werden. `flag[i]` zeigt an, dass P_i in seinen kritischen Bereich möchte, und kann von P_j abgefragt werden. Initialisiert wird `flag[i]` mit `FALSE`. Der Algorithmus für P_i lautet in diesem Fall:

PROGRAMM

Bessere aber immer noch fehlerhafte Variante

```
flag[i] := FALSE;
repeat
  Unkritischer Bereich;
  flag[i] := TRUE;          (* A *)
  while flag[j] do          (* B *)
    noop;                   (* B *)
  Kritischer Bereich;
  flag[i] := FALSE;
until FALSE;
```

Auch diese Lösung ist leider nicht haltbar, da es hier ebenfalls zu zeitlichen Abläufen kommen kann, die zu einem kompletten Stillstand des Systems führen. Einfachstes Beispiel:

BEISPIEL

Gegenbeispiel zur Korrektheit obiger Variante

Zeit	P_i, P_j
1	`flag[i] := TRUE;`
2	`flag[j] := TRUE;`
3	`while flag[i] do noop;`
4	`while flag[j] do noop;`

An dem erreichten Zustand wird sich in alle Ewigkeit nichts mehr ändern. Man kann sich leicht selbst davon überzeugen, dass auch ein Vertauschen der Reihenfolge von A und B die Situation nicht verbessert.

In Abbildung 3.5 ist eine Kombination der beiden Algorithmen zugrundeliegenden Ideen dargestellt, die zu einer Lösung des wechselseitigen Ausschlussproblems führt, für die alle drei Bedingungen erfüllt sind.

Oder etwas formaler (für Prozess P_i):

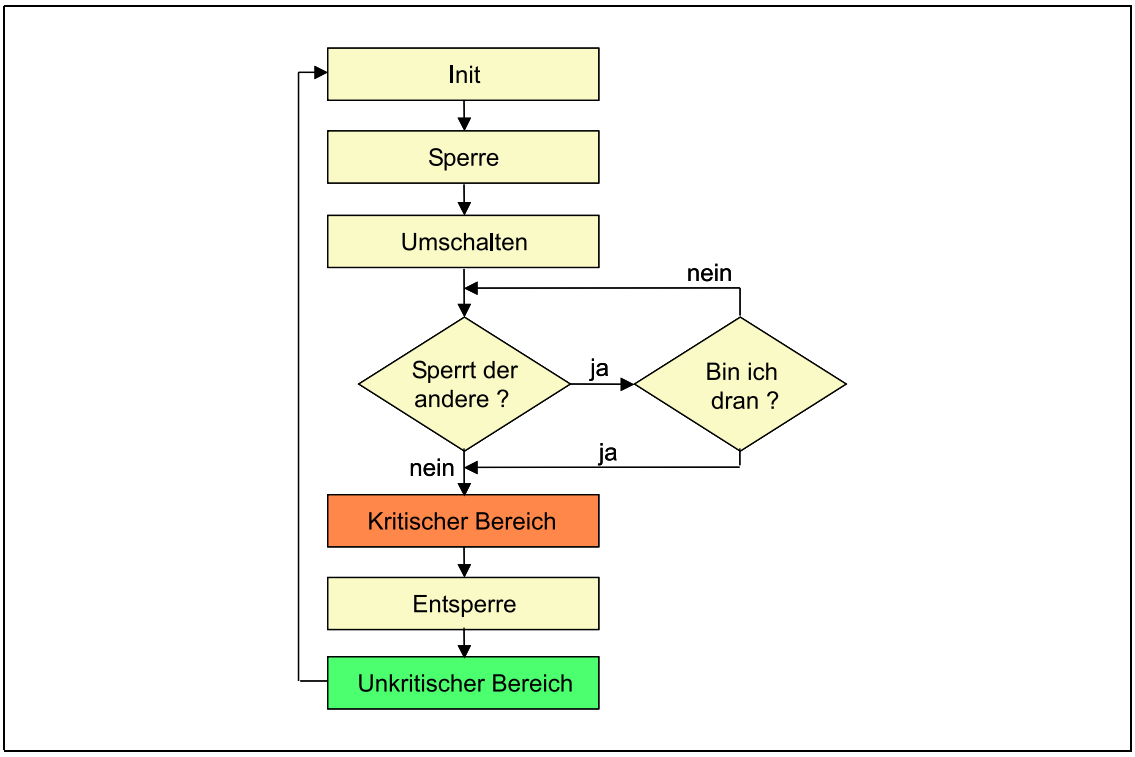


Abbildung 3.5: Korrekte Lösung für das wechselseitige Ausschlussproblem

PROGRAMM

Korrekte Lösung

```
repeat
  flag[i] := TRUE;
  turn := j;
  while (flag[j] and turn = j) do
    noop;
  Kritischer Bereich;
  flag[i] := FALSE;
  Unkritischer Bereich;
until FALSE;
```

Warum ist diese Lösung korrekt?

1. Mutual-Exclusion (höchstens ein Prozess im kritischen Bereich)

   P_i kann genau dann seinen kritischen Bereich betreten, wenn die Bedingung `(flag[j] and turn = j)` nicht erfüllt ist. Also ist `flag[j] = FALSE` oder `turn = i`, d.h. entweder möchte P_j nicht oder P_j ist nicht dran. P_j kann jedoch auf keinen Fall seine While-Schleife überwunden haben und befindet sich mithin nicht in seinem kritischen Bereich.

2. Progress (Eintritt eines Prozesses nach endlicher Zeit)

   1. Fall: Beide Prozesse möchten in den KB. Dann hängt die Entscheidung nur von der Bedingung `turn = i` bzw. `turn = j` ab. Dies ist auf jeden Fall für i oder j erfüllt, d.h. einer der beiden Prozesse betritt den KB nach endlicher Zeit.
   2. Fall: Nur P_i möchte in den KB (P_j analog). P_i setzt `flag[i] := TRUE` und `turn`

`:= j`. Wegen der Und-Verknüpfung in der While-Schleife und der Tatsache, dass `flag[j] = FALSE` ist, folgt, dass P_i den KB nach endlicher Zeit betritt.

3. Bounded-Waiting (nur endlich oft Vortritt für andere Prozesse)

   P_i kann maximal einmal überholt werden. Sobald nämlich P_i in der While-Schleife angekommen ist, muss er nur dann warten, wenn `flag[j] = TRUE` und

   (a) P_j noch vor `turn := i` ist oder

   (b) P_j `turn := i` passiert hat, bevor P_i `turn := j` gesetzt hat.

   Da P_j nur im UKB sterben darf, wird er im ersten Fall nach endlicher Zeit `turn := i` bzw. im zweiten Fall `flag[j] := FALSE` setzen. Von da an ist die Schleifenbedingung von P_i aber nicht mehr erfüllt, und sobald P_i diese Bedingung abgefragt hat, kann er in seinen kritischen Bereich eintreten.

Eine Erweiterung dieses Algorithmus für $n > 2$ Prozesse $P_1, P_2, ..., P_n$ stellt sich als keineswegs trivial heraus, wenn weiterhin nur einfache Lese-/Schreiboperationen in atomarer Implementierung zur Verfügung stehen.

3.2.2 Der Bakery-Algorithmus

Einen wesentlichen Fortschritt stellt der so genannte Bakery-Algorithmus dar. Im Alltag kann man dem hier realisierten Prinzip des Öfteren begegnen, beispielsweise in der Zahnarztpraxis, auf dem Einwohnermeldeamt oder eben (vereinfacht) in Bäckereien.

Der Algorithmus beruht auf der Ausgabe von Nummern für wartende Kunden. Solange keiner wartet, steht der Ticketzähler auf Null. Kommt der erste Interessent an, zieht er die Nummer 0. Später ankommende Kunden lösen weitere Tickets mit jeweils höheren Nummern. Es ist allerdings nicht ausgeschlossen, dass die gleiche Nummer an mehrere gleichzeitig ankommende Kunden vergeben wird. Die Bedienung erfolgt dann nach folgender Strategie: Die kleinste Nummer ist jeweils als nächste dran. Falls sie an mehrere Kunden vergeben wurde, kommt derjenige dran, dessen Name im Alphabet am weitesten vorne steht bzw. dessen Prozessnummer am niedrigsten ist (Tie-Break).

Als kritischen Bereich kann man leicht das Programmstück ausmachen, das die Ziehung der Nummern regelt. Im Algorithmus verwendet man hier ein Variablenfeld `choosing[i]`, das für jedes i mit `FALSE` vorbesetzt ist. Setzt Prozess P_i seinen Feldwert `choosing[i] := TRUE`, so signalisiert er damit, dass er eine Nummer ziehen will (d.h. auf gemeinsame Daten zugreifen möchte). Der Algorithmus stellt sicher, dass diese gemeinsamen Daten während des Zugriffs von P_i nicht von anderen Prozessen verändert werden, da die anderen ihrerseits vorher `choosing[i] = TRUE` abfragen müssen. Nach erfolgter Nummernziehung setzt P_i seinen Feldeintrag durch `choosing[i] := FALSE` zurück.

Vereinbart man nun noch, dass

`(a,b) < (c,d)` $\iff$ `entweder a < c oder (a = c und b < d)`

gilt, so lautet der Algorithmus (für den i-ten von n Prozessen) wie folgt:

PROGRAMM

Algorithmus für Prozess P_i

```
repeat
  choosing[i] := TRUE;
  number[i] := max(number[0], number[1], ..., number[n-1]) + 1;
  choosing[i] := FALSE;
  for j := 0 to n-1 do
  begin
    while choosing[j] do
      noop;
    while number[j] <> 0 and (number[j], j) < (number[i], i) do
      noop;
  end;

  Kritischer Bereich;

  number[i] := 0;

  Unkritischer Bereich;
until FALSE;
```

Die Korrektheit dieser Lösung soll hier nicht im Detail nachgewiesen werden. Dass jeweils höchstens einer der Prozesse im kritischen Bereich ist, wird durch die oben dargelegte Strategie (einschließlich der Tie-Break-Regelung) gewährleistet. Da die Abarbeitung der Schlange im Wesentlichen nach FIFO geschieht, ist auch Progress und Bounded-Waiting gewährleistet. Zwei kleine Schönheitsfehler könnte man dieser Lösung ankreiden: Einmal ist der Algorithmus nicht ganz fair, da Prozesse mit kleinen Nummern einen winzigen Vorteil erhalten. Außerdem könnte man sich Szenarien ausdenken, in denen der Algorithmus bei Verwendung einer beschränkten Ganzzahl-Arithmetik versagt.

3.2.3 Enqueue und Dequeue

Eine Variante des Bakery-Algorithmus verwendet die atomaren Operationen **enqueue** und **dequeue** zur Verwaltung einer FIFO-Warteschlange von n Prozessen $P_1, ..., P_n$. Die Warteschlange kann man sich als verkettete Liste von Prozessnummern vorstellen, die in einem Array `F[0..n]` verwaltet wird. `F[0]` ist dabei die Nummer des Prozesses, der sich aktuell in seinem kritischen Bereich befindet, `F[F[0]]` ist die Nummer des ältesten (d.h. des am längsten auf den KB-Eintritt wartenden) Prozesses usw. Wartet kein Prozess, so gilt `F[0] = 0`. Die Arbeitsweise der atomaren Operationen enqueue und dequeue und ein Beispiel dazu ist in Abbildung 3.6 abgebildet.

`Enqueue(i)` ordnet also den Prozess P_i in die Warteschlange an letzter Position ein, während `dequeue(j)` den ältesten Prozess P_j aus der Warteschlange eliminiert. `p` ist dabei jeweils ein Zeiger auf den Platz im Array `F`, dessen Nummer der Nummer des bislang jüngsten Prozesses entspricht. Die Bedingung `p`$\neq$`i` überprüft, ob sich hinter P_i noch weitere Prozesse in der Warteschlange befinden.

Der Algorithmus für P_i hat dann folgende Gestalt:

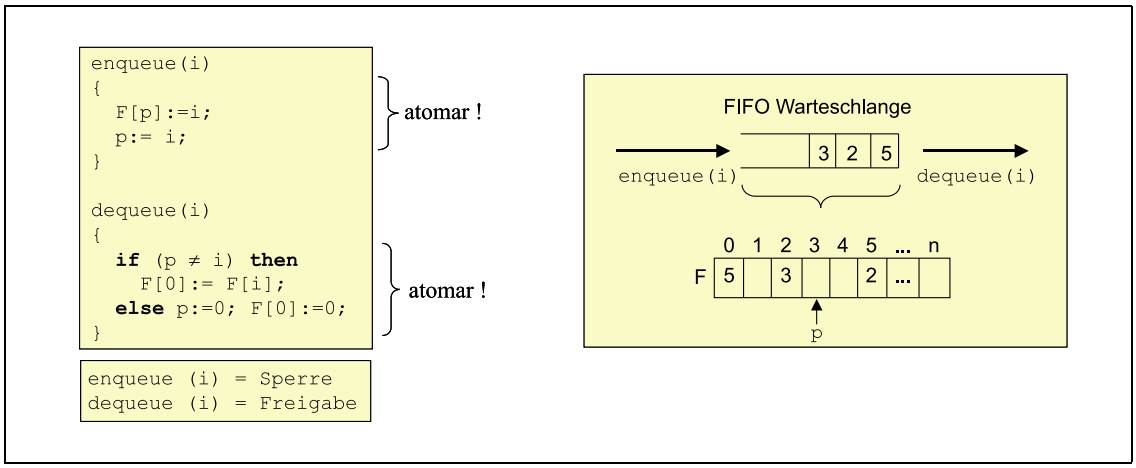


Abbildung 3.6: Arbeitsweise von enqueue und dequeue

PROGRAMM

Algorithmus für P_i

```
repeat
  Unkritischer Bereich;
  enqueue(i);
  while F[0] <> i do
    noop;
  Kritischer Bereich;
  dequeue(i);
until FALSE;
```

Sind `enqueue` oder `dequeue` nicht atomar, kann es leicht zu Schwierigkeiten kommen, wie wir an dem Beispiel in Abbildung 3.7 zeigen wollen. Sei hierfür `enqueue` teilbar und `p` mit 0 initialisiert. Wir betrachten die Prozesse P_j und P_k:

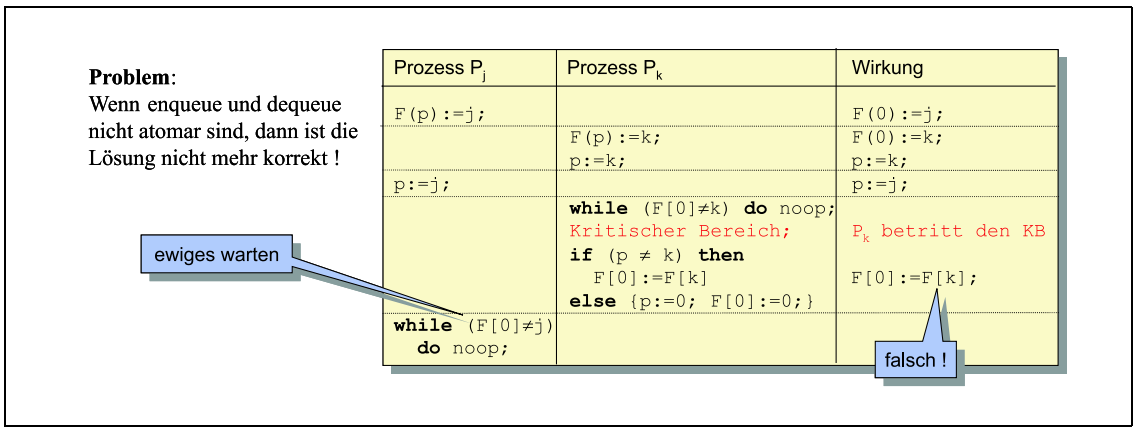


Abbildung 3.7: Problematik bei teilbaren enqueue- und dequeue-Operationen

3.2.4 Synchronisationsmechanismen mit atomaren Operationen

Interrupt

Am einfachsten lassen sich atomare Operationen über eine Interrupt-Steuerung realisieren. Das Programmstück, das atomar sein soll, wird dabei eingeleitet von einem Befehl, der

einen Interrupt unmöglich macht, und schließt mit einem Befehl, der Interrupts wieder ermöglicht (siehe Abbildung 3.8).

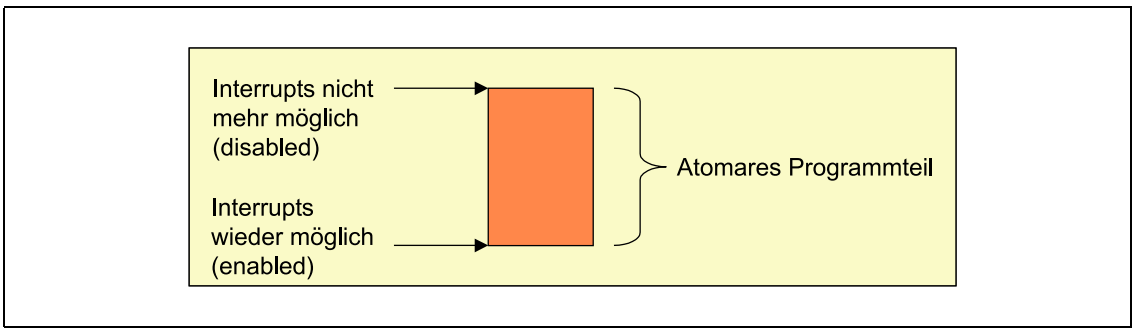


Abbildung 3.8: Kritische Bereiche und Interrupts

Dieses Verfahren stellt eine korrekte Lösung dar, da wir sicher sein können, dass ein Prozess, der in seinem kritischen Bereich ist, nicht gestört werden kann. Es ist allerdings nur für Einprozessorsysteme brauchbar.

Test-and-Set

Eine von vielen Computersystemen hardwaremäßig angebotene Möglichkeit besteht darin, eine Boole'sche Variable auf ihren Inhalt hin zu testen und diesen gegebenenfalls zu modifizieren, und zwar innerhalb einer unteilbaren Operation. Eine derartige Test-and-Set-Instruktion hat typischerweise folgendes Aussehen:

PROGRAMM

Test-and-Set

```
function Test-and-Set(var target: boolean): boolean;
begin
  Test-and-Set := target;
  target := TRUE;
end;
```

Diese ganze Operation erfolgt atomar. Das bedeutet, dass im Falle des gleichzeitigen Aufrufs zweier Test-and-Set-Operationen auf verschiedenen CPUs letztlich eine (zeitlich) sequenzielle Abarbeitung in beliebiger Reihenfolge stattfindet.

Der Test-and-Set-Mechanismus kann sehr gut zur Lösung des wechselseitigen Ausschlussproblems verwendet werden. Hierzu deklariert man eine globale Boole'sche Variable `lock`, initialisiert sie einmalig mit `FALSE` und verwendet für jeden Prozess den nachfolgenden Algorithmus:

PROGRAMM

Lösung mit Test-and-Set

```
repeat
  while Test-and-Set(lock) do
    noop;
  Kritischer Bereich;
  lock := FALSE;
  Unkritischer Bereich;
until FALSE;
```

Es ist gewährleistet, dass immer höchstens ein Prozess in seinem kritischen Bereich ist. Allerdings ist die Bounded-Waiting-Bedingung nicht erfüllt, da ein Prozess beliebig oft

Pech haben kann, weil er die Variable `lock` jedesmal erst abfragt, nachdem ihm schon wieder ein anderer Prozess zuvorgekommen ist. Um dies zu vermeiden, bedarf es eines komplizierteren Algorithmus, der die Prozesse in zyklischer Reihenfolge testet und den wir hier für Prozess P_i ohne nähere Erläuterung angeben:

PROGRAMM

Lösung mit Test-and-Set, welche die Boundet-Waiting-Bedingung erfüllt

```
var j: 0..n-1;
    key: boolean;

repeat
  waiting[i] := TRUE;
  key := TRUE;
  while waiting[i] and key do
    key := Test-and-Set(lock);
  waiting[i] := FALSE;
  Kritischer Bereich;

  j := i+1 mod n;
  while (j <> i) and not waiting[j] do
    j := j+1 mod n;

  if j = i then
    lock := FALSE
  else
    waiting[j] := FALSE;

  Unkritischer Bereich;
until FALSE;
```

Swap

Ein anderer oftmals implementierter Mechanismus erlaubt das atomare Vertauschen des Inhalts zweier Boole'scher Variablen. Diese Swap-Operation sieht folgendermaßen aus:

PROGRAMM

Swap

```
procedure Swap (var a, b: boolean);
var temp: boolean;
begin
  temp := a;
  a := b;
  b := temp;
end;
```

Man nutzt sie ähnlich wie Test-and-Set zum atomaren Aufschließen eines globalen Schlosses `lock` mit einem lokalen Schlüssel `key`, beides Boole'sche Variablen. `lock` wird zu Beginn auf `FALSE` gesetzt, um anzuzeigen, dass sich kein Prozess in seinem kritischen Bereich befindet. Ein Prozess P_i, der in seinen kritischen Bereich möchte, setzt seinen `key` auf `TRUE` und beginnt, den Inhalt von `key` und `lock` mittels `swap` zu vertauschen, um festzustellen, ob der Zugang zum kritischen Bereich gerade möglich ist. Solange ein anderer Prozess P_k in seinem kritischen Bereich ist, gilt `lock = TRUE`, und die Vertauschungsaktionen von P_i ändern nichts am Status quo. Verlässt P_k seinen kritischen Bereich, so setzt er `lock` auf `FALSE`. Bei der nächsten Vertauschungsaktion merkt P_i dann, dass er in seinen kritischen Bereich kann, und hat zugleich nach außen bekanntgegeben, dass für andere wartende

Prozesse der Zugang jetzt wieder gesperrt ist:

PROGRAMM

Lösung mit Swap

```
repeat
  key := TRUE;
  repeat
    Swap(lock, key);
  until key = FALSE;

  Kritischer Bereich;
  lock := FALSE;
  Unkritischer Bereich;
until FALSE;
```

Die grundsätzliche Idee ist praktisch dieselbe wie beim Test-and-Set-Vorschlag. Daher verwundert es nicht, dass auch der Swap-Algorithmus kein Bounded-Waiting garantiert. Im Übrigen ist global jeweils nur bekannt, dass ein Prozess im kritischen Bereich ist, man kann aber nicht feststellen, welcher es ist.

3.3 Semaphore

3.3.1 Das Konzept eines Semaphors

Die bisher vorgestellten Software-Lösungen für das wechselseitige Ausschlussproblem basieren auf elementaren atomaren Operationen (wie Lese-/Schreibzugriffen, Test-and-Set, Swap) und werden bei der Übertragung auf komplexere Aufgabenstellungen sehr schnell unübersichtlich und schwer zu verifizieren. Daher liegt es nahe, höhersprachliche Konstrukte zu entwickeln, die ein größeres Maß an Übersichtlichkeit ermöglichen und dadurch weniger schwer zu verstehen und vor allem weniger fehleranfällig sind. Realisiert werden sie ihrerseits dann durch einfache atomare Konzepte, wie wir sie bisher kennengelernt haben.

Ein erstes wichtiges Beispiel eines solchen komplexeren Hilfsmittels finden wir im Konzept der Semaphoren. Veranschaulichen kann man sich einen Semaphor beispielsweise anhand eines Aachener Parkhauses, das an seinem Eingang einen Zähler besitzt. Am frühen Morgen wird der Zähler initialisiert und zeigt die Anzahl der verfügbaren Parkplätze an. Sobald ein Auto in das Parkhaus hineinfährt, wird der Zähler am Eingang um eins verringert. Verlässt ein Wagen das Parkhaus, wird der Zähler wieder um eins erhöht. Steht der Zähler auf 0, muss ein ankommendes Fahrzeug solange um den Block fahren, bis ein Auto das Parkhaus verlässt und dem Zähler signalisiert hat, dass jetzt wieder ein freier Platz zur Verfügung steht. Das Initialisieren, Inkrementieren und Dekrementieren des Zählers erfolgt dabei stets atomar.

Eine Abstraktion dieses Mechanismus führt zum Semaphorkonzept. Ein Semaphor `S` ist demnach eine Integervariable, die nur durch drei atomare Operationen namens `init`, `wait` und `signal` verändert werden kann. In den meisten Fällen ist mit einem Semaphor außerdem eine **assoziierte Warteschlange** verknüpft, um auszuschließen, dass Prozesse über längere Zeit in einer Abfrageschleife verweilen und somit unnötigerweise die CPU belegen (bzw. in der automobilistischen Variante ständig um den Block fahren, anstatt den Motor abzustellen und vor dem Eingang auf das Freiwerden des nächsten Parkplatzes zu warten). Assoziierte Warteschlangen verhindern ein derartiges **Busy-Waiting**, indem sie die Prozessnummern in der Reihenfolge, in der sie dran sind, festhalten, woraufhin sich

die wartenden Prozesse schlafen legen können und daher einstweilen keine CPU-Zeit mehr beanspruchen. Jedesmal, wenn der kritische Bereich wieder frei wird, lässt sich der Warteschlange entnehmen, welcher Prozess als nächster an die Reihe kommt und demzufolge zu wecken ist.

Nun zur Definition der drei atomaren Operationen:

1. `init(S, Anfangswert)`

   Die Wirkung dieser Operation ist `S := Anfangswert`. Hierdurch wird also der Semaphor `S` initialisiert. Als Anfangswert nimmt man sehr oft die Anzahl von Prozessen, die sich gleichzeitig in einem kritischen Bereich aufhalten dürfen (in unserem Beispiel etwa die Anzahl der anfangs freien Parkplätze). Beim wechselseitigen Ausschlussproblem ist die Anzahl der Prozesse im kritischen Bereich auf höchstens 1 begrenzt, daher initialisiert man hier mittels `init(S,1)`. Einen derartigen Semaphor, der nur die Werte 0 oder 1 annehmen kann, nennt man übrigens **binär** im Gegensatz zu einem **Zählsemaphor**, dessen Wertebereich nicht nach oben beschränkt ist.

2. `wait(S)` (historisch auch P(S) von niederl. „proberen“)

   Die Wirkung dieser Instruktion entspricht

   PROGRAMM

   Wirkung von wait

   ```
   wait(S)
   {
     while S <= 0 do
       noop;
     S := S-1;
   }
   ```

   Man muss sich dabei unbedingt ins Gedächtnis rufen, dass diese ganze Operation atomar erfolgt. Ist mit `S` eine Warteschlange assoziiert, so lautet der atomar implementierte Algorithmus:

   PROGRAMM

   Wait mit assoziierter Warteschlange

   ```
   wait(S)
   {
     S := S-1;
     if S < 0 then
       ordne Prozess in Warteschlange an Position -S ein;
   }
   ```

3. `signal(S)` (oder auch V(S) von niederl. „verhogen“ = erhoehen)

   Diese letzte Semaphoroperation bewirkt

   PROGRAMM

   Wirkung von signal

   ```
   signal(S)
   {
     S := S+1
   }
   ```

bzw. mit assoziierter Warteschlange

PROGRAMM

Signal mit assoziierter Warteschlange

```
signal(S)
{
  if S <= 0 then
    wecke den ältesten wartenden Prozess auf
    und schicke ihn in den kritischen Bereich;
}
```

Beides ist wiederum atomar zu realisieren.

Hat man einen Semaphor `S` samt der drei angegebenen Operationen zur Verfügung, kann man folgenden prinzipiellen Lösungsalgorithmus für das wechselseitige Ausschlussproblem angeben:

PROGRAMM

Lösung mit Semaphoren

```
globale Initialisierung:

init(S,1);

Algorithmus für Prozess i :

repeat
  wait(S);
  Kritischer Bereich;
  signal(S);
  Unkritischer Bereich;
until FALSE;
```

Dies gilt in direkter Analogie zum Parkhausbeispiel (wenn man von der etwas unrealistischen Voraussetzung ausgeht, das Parkhaus bestehe aus genau einem Parkplatz).

Semaphore lassen sich auch anderweitig einsetzen, etwa im Bereich der Ablaufsteuerung: Hat man zwei Prozesse P_0 und P_1 und will gewährleisten, dass P_1 vor P_0 ausgeführt wird, lässt sich dies einfach erreichen, indem man einen Semaphor `a` benutzt, ihn mit 0 initialisiert sowie am Anfang von P_0 den Befehl `wait(a)` und am Ende von P_1 den Befehl `signal(a)` einfügt. Diese umgekehrte Semaphornutzung bewirkt, dass P_0 solange warten muss, bis P_1 fertig ist.

Vor allem verwendet man Semaphoren zur Lösung von Koordinationsaufgaben wie dem bereits erwähnten Erzeuger-Verbraucher-Problem, den Reader-Writer-Problemen, dem Fünf-Philosophen-Problem und einer ganzen Reihe verwandter Aufgaben.

3.3.2 Das Erzeuger-Verbraucher-Problem

Betrachten wir zunächst noch einmal das Erzeuger-Verbraucher-Problem für den Fall von n Erzeugern, m Verbrauchern und einem Zwischenlager der Größe MAX. Die Erzeugerprozesse produzieren und legen im Lager ab, solange der dortige Bestand weniger als MAX beträgt. Die Verbraucherprozesse entnehmen ihrerseits Elemente aus dem Lager, solange der Bestand größer als 0 ist, und konsumieren. Teilt man die jeweiligen Prozesse auf, ergibt

sich beim Erzeuger das Produzieren und beim Verbraucher das Konsumieren als unkritischer Bereich, während das Ablegen bzw. Entnehmen als kritischer Bereich anzusehen ist, da Erzeuger wie Verbraucher dabei auf gemeinsame Daten zugreifen müssen.

Neu an der im Folgenden vorgestellten Lösung ist, dass der exklusive Zugriff auf das Lager mittels eines Semaphors `s` realisiert werden soll. Initialisiert wird `s` durch `init(s,1)`. Der Sperrmechanismus zur Sicherung des exklusiven Zugriffs sieht schematisch wie folgt aus:

PROGRAMM

Sperrmechanismus zum exklusiven Zugriff

```
...
wait(s);
  Kritischer Bereich;
signal(s);
...
```

Außerdem lassen sich die beiden Nebenbedingungen (dass zum einen in ein volles Lager nichts gelegt und zum anderen aus einem leeren Lager nichts entnommen werden kann) über zwei zusätzliche Zählsemaphoren `f` und `c` verwirklichen. Dabei bedeutet

$$f = 0 \quad \Leftrightarrow \quad \text{Lager ist leer}$$
$$c = 0 \quad \Leftrightarrow \quad \text{Lager ist voll}$$

Die Initialisierung geschieht mittels `init(f,0)` und `init(c,MAX)`. Während der gesamten Laufzeit soll dann $f + c$ invariant bleiben, d.h. eine Inkrementierung von `f` bedingt eine Dekrementierung von `c` und vice versa. Die Algorithmen lauten:

PROGRAMM

Erzeuger

```
repeat
  produziere ein Element und lege es in nextp ab;
  wait(c);
  wait(s);
  füge nextp in den Puffer ein;
  signal(s);
  signal(f);
until FALSE;
```

PROGRAMM

Verbraucher

```
repeat
  wait(f);
  wait(s);
  entferne Element aus dem Puffer und lege es unter nextc ab;
  signal(s);
  signal(c);
  verbrauche das Element in nextc;
until FALSE;
```

Bemerkenswert ist, dass eine Vertauschung der `signal`-Instruktionen nichts an der Korrektheit ändert, wohl aber ein Vertauschen der `wait`-Befehle.

3.3.3 Die Reader-Writer-Probleme

Das nächste klassische Problem taucht vor allem im Zusammenhang mit der Verwaltung von Datenbanken auf. Hierbei haben zwei Arten von Prozessen Zugriff auf ein gemeinsames Datenobjekt (z.B. eine Datei): Prozesse der ersten Sorte (die **Writer**) machen Updates der Daten. Dabei muss sichergestellt werden, dass ihr Schreibzugriff exklusiv erfolgt, d.h. kein anderer Prozess (egal ob Writer oder Reader) darf während des Updates aktiv sein. Die zweite Art von Prozessen (die **Reader**) stellen Anfragen, die simultan stattfinden können. Dies kann man sich beispielsweise an einem Zugfahrplan veranschaulichen, der am Bahnhof aushängt: Ein Fahrplanwechsel (also ein Update) wird exklusiv durchgeführt (d.h. ein einzelner Bahnbeamter wechselt den aushängenden Fahrplan in einem Augenblick, in dem ihn niemand konsultiert), während eine prinzipiell beliebige Anzahl von Interessenten gleichzeitig im aushängenden Plan nach Abfahrtszeiten, Gleisnummern etc. suchen kann.

Man unterscheidet nun drei Varianten des Reader-/Writer-Problems (nach ihrem Erfinder P.J. Courtois auch **Courtois-Probleme** genannt), die sich hinsichtlich ihrer Prioritätsregelungen unterscheiden:

Erstes R/W-Problem: Kein Reader muss auf eine Eintrittserlaubnis warten, es sei denn, es ist gerade ein Writer in seinem kritischen Bereich. In diesem Fall kann es geschehen, dass die Reader das System monopolisieren, d.h. ein Writer, der gerne ein Update machen möchte, kann von einem Verschwörerkreis aus Readern auf ewig vom Zugang abgehalten werden (**Starvation-Problem**).

Zweites R/W-Problem: Man kann versuchen, diesem Nachteil zu begegnen, indem man neu hinzukommenden Lesewilligen den Zutritt untersagt, sobald ein Writer wartet. Jetzt können jedoch die Writer das Monopol ausüben, sodass keinem Reader mehr der Zutritt in seinen kritischen Bereich gelingt.

Drittes R/W-Problem: Um Fairness zu gewährleisten, werden Lese- und Schreibphasen alterniert: Ist gerade Lesephase und meldet sich ein Writer an, werden keine neuen Reader mehr zugelassen. Sobald alle Reader fertig sind, darf der Writer zugreifen. Ist andererseits gerade eine Schreibphase und meldet sich ein Reader an, wird das Ende dieser Schreibphase abgewartet und dann eine Lesephase gestartet.

Im Folgenden soll eine Lösung für das erste R/W-Problem angegeben werden, welche die folgenden Semaphore und Zählvariablen verwendet:

Semaphor `s`	:	Schreibphase
Semaphor `w`	:	Lesephase
Zählvariable `n`	:	Anzahl der gleichzeitig aktiven Leser

Der Semaphor `s` wird durch `init(s,1)` initialisiert und sowohl von Reader- als auch von Writer-Prozessen verwendet, um Writern einen exklusiven Zugriff zu ermöglichen. Der Semaphor `w` (ebenfalls mit `init(w,1)` initialisiert) soll sicherstellen, dass ein Update von `n` ungestört erfolgen kann. Die Zählvariable `n` wird anfangs mit 0 initialisiert.

PROGRAMM

Writer

```
repeat
  wait(s);
  schreibe;
  signal(s);
until FALSE;
```

PROGRAMM

Reader

```
repeat
  wait(w);
  n := n + 1;
  if n = 1 then
    wait(s);
  signal(w);
  lies;
  wait(w);
  n := n - 1;
  if n = 0 then
    signal(s);
  signal(w);
until FALSE;
```

Wenn sich hierbei ein Writer in seinem kritischen Bereich aufhält und m Reader warten, so wartet einer davon in der zu `s` assoziierten Warteschlange, während $m-1$ in der zu `w` gehörenden Schlange eingereiht werden.

3.3.4 Das Fünf-Philosophen-Problem

Dies ist ein weiterer Klassiker, dessen Berühmtheit allerdings weniger auf etwaige praktische Relevanz zurückzuführen ist, sondern eher darauf, dass man an diesem Problem besonders schön das Zusammenspiel von Prozessen und die Gefahr von Verklemmungen darstellen kann. Hier zunächst eine anschauliche Darstellung des Sachverhalts.

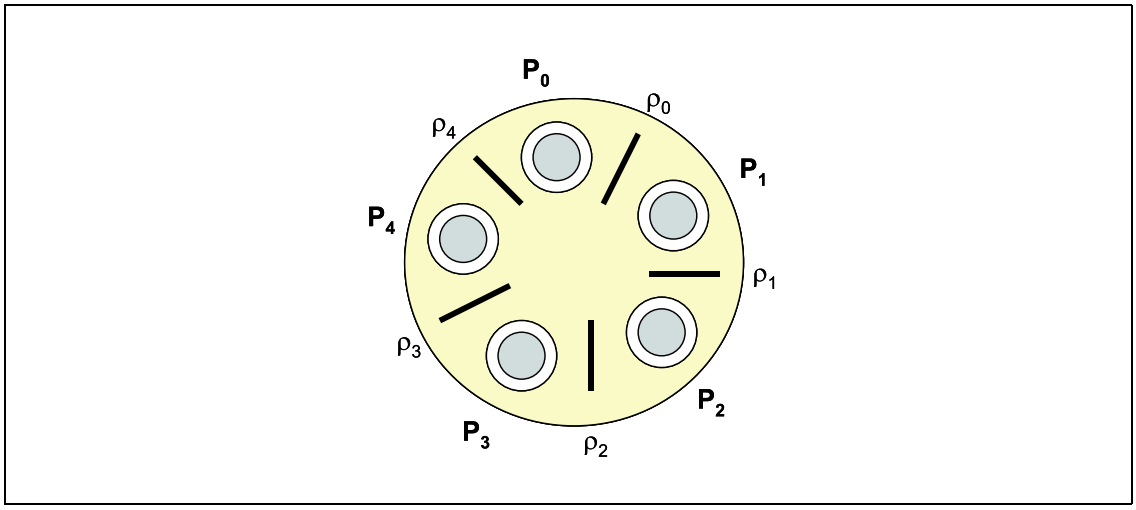


Abbildung 3.9: Das Fünf-Philosophen-Problem

Philosophen verbringen (zugegebenermaßen leicht abstrakt dargestellt) ihr Leben mit Denken und Essen. Betrachten wir also fünf östliche Exemplare davon, die sich um einen runden Tisch versammelt haben (siehe Abbildung 3.9). Vor jedem von ihnen befindet sich ein Teller voll Reis, und zwischen zwei Tellern liegt jeweils eines der aus Chinarestaurants hinreichend bekannten Essstäbchen. Essen kann auch ein Philosoph nur unter Zuhilfenahme zweier dieser Stäbchen. Von Zeit zu Zeit nun wird einer unserer Helden hungrig und greift nacheinander nach den beiden Stäbchen, die rechts und links von seinem Teller liegen. Natürlich kann er ein Stäbchen nur dann in die Hand nehmen, wenn es gerade nicht vom

betreffenden Nachbarn benutzt wird. Ist es dem hungrigen Philosophen geglückt, beide Stäbchen an sich zu bringen, so isst er, ohne sie jemals loszulassen, bis er satt ist. Anschließend legt er sie an ihren ursprünglichen Platz zurücklegt und verfällt von neuem ins Grübeln.

Abstrakt gesehen haben wir also fünf Prozesse (Philosophen) $P_0, ..., P_4$ vor uns, die zyklisch angeordnet sind. Ferner sind noch fünf Betriebsmittel (Stäbchen) $s_0, ..., s_4$ gegeben. Jeder Prozess P_i benötigt zeitweise zwei Betriebsmittel s_i (linkes Stäbchen) und $s_{(i+1)mod5}$ (rechtes Stäbchen) gleichzeitig.

Schematisch haben die Prozesse folgende Struktur:

PROGRAMM

Schematische Struktur

```
repeat
  Unkritischer Bereich ('Denken')
  Kritischer Bereich   ('Essen': Pi braucht s[i] und s[(i+1) mod 5])
until FALSE;
```

Naive Lösung

Am naheliegendsten ist es wohl, jedes Stäbchen durch einen Semaphor `s[0]` ... `s[4]` zu repräsentieren, der jeweils durch `init(s[i],1)` initialisiert wird. Prozess P_i arbeitet dann gemäß folgendem Algorithmus:

PROGRAMM

Naive Lösung

```
repeat
  wait(s[i]);
  wait(s[i+1 mod 5]);
  Essen;
  signal(s[i]);
  signal(s[i+1 mod 5]);
  Denken;
until FALSE;
```

Aber: Bei dieser Lösung sind Verklemmungen (**Deadlocks**) nicht ausgeschlossen. Eine Verklemmung tritt ein, wenn alle Prozesse annähernd gleichzeitig in ihren kritischen Bereich wollen. Jeder Philosoph greift dann (wahre Philosophen sind in aller Regel Linkshänder!) zunächst nach seinem linken Stäbchen und wartet mit diesem in der Hand darauf, dass er das rechte Stäbchen aufnehmen kann. Da dies jeder der fünf Philosophen tut, steht zu befürchten, dass sie bis in alle Ewigkeit so dasitzen und sich gegenseitig blockieren.

Lösungsmöglichkeiten zur Vermeidung von Deadlocks

Man kann das Eintreten eines Deadlocks durch die Einführung von Zwischenzuständen verhindern (siehe Abbildung 3.10).

Der Zustand „essen“ bleibt weiterhin kritisch, ebenso symbolisiert der Zustand „denken“ auch hier den unkritischen Bereich. Neu ist ein Zustand „hungrig“, den ein Philosoph einnimmt, wenn er sich zum Essen anmeldet.

Die Lösung basiert dann auf dem Semaphor `s` (initialisiert mit `init(s,1)`) sowie den Semaphoren `h[0]`, ..., `h[4]` (jeweils mit `init(h[i],0)` initialisiert). Dabei ist `h[i] = 1` gleichbedeutend damit, dass beide Stäbchen für P_i frei sind und P_i obendrein hungrig

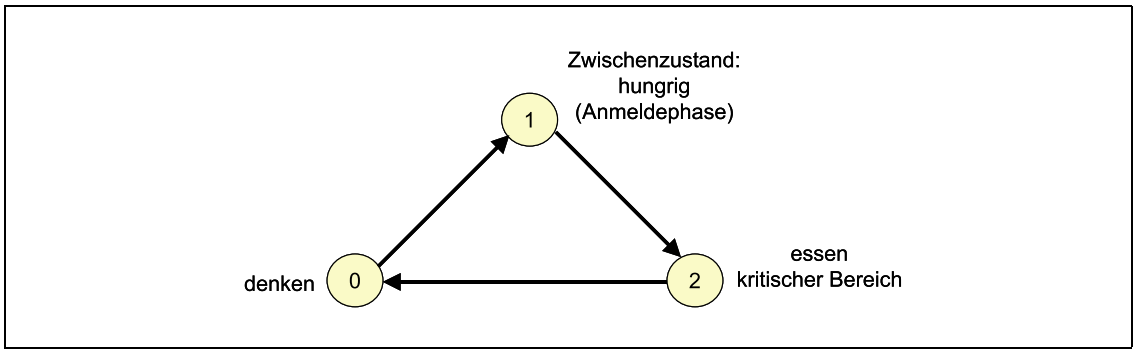


Abbildung 3.10: Die drei (einzig) möglichen Zustände eines Philosophen

ist. Die Zustände der Philosophen werden mittels der Kontrollvariablen `c[i]` festgehalten, wobei `c[i]` die Werte 0, 1 oder 2 (für „denken“, „hungrig“ bzw. „essen“) annehmen kann. Des Weiteren wird die folgende Testprozedur benötigt:

PROGRAMM

Testprozedur

```
procedure test(k)

if(c[(k-1) mod 5] <> 2     (* linker Nachbar isst nicht  *)
  and c[k] = 1             (* ich bin hungrig            *)
  and c[(k+1) mod 5] <> 2) (* rechter Nachbar isst nicht *)
then begin
  c[k] := 2;
  signal(h[k]);
end;
```

Der Algorithmus für den Philosophen P_i lautet damit:

PROGRAMM

Algorithmus für Philosoph i

```
repeat
  Denken;
  wait(s);
    c[i] := 1;          (* hungrig werden                *)
    test(i);            (* Test, ob Stäbchen frei        *)
  signal(s);
  wait(h[i]);
  Essen;
  wait(s);
    c[i] := 0;          (* zurück zum Denken             *)
    test(i+1 mod 5);    (* Test, ob Nachbarn essen können *)
    test(i-1 mod 5);
  signal(s);
until FALSE;
```

Bei dieser Lösung ist das Eintreten eines Deadlocks ausgeschlossen. Allerdings kann es immer noch vorkommen, dass einzelne Philosophen verhungern, d.h. es ist möglich, dass ein Prozess auf unbestimmte Zeit blockiert wird. Betrachten wir z.B. den hungrigen Prozess P_2. P_1 und P_3 können sich dergestalt gegen ihn verschwören, dass zunächst P_1 mit den Stäbchen s_1 und s_2 isst. Kurz bevor P_1 sein Mahl beendet, beginnt P_3 mit den Stäbchen

s_3 und s_4 zu essen. In ähnlicher Weise kurzzeitig überlappend startet dann wieder P_1 seine Essphase usw. bis zum bitteren Ende von P_2.

Eine andere Möglichkeit, gegen die Gefahr eines Deadlocks vorzugehen, besteht in der Einführung einer Asymmetrie unter den Prozessen, indem man einen Philosophen (etwa P_4) einfach zum Rechtshänder deklariert. Für die Algorithmen bedeutet das:

P_0 ... P_3 (Linkshänder)	P_4 (Rechtshänder)
`wait(s[i])`	`wait(s[0])`
`wait(s[i+1])`	`wait(s[4])`
Essen	Essen
`signal(s[i])`	`signal(s[0])`
`signal(s[i+1])`	`signal(s[4])`

Warum ist dieser Vorschlag deadlockfrei? Wie wir später noch ausführlich sehen werden, wenn wir zur eigentlichen Besprechung von Deadlocks kommen, ist für eine Verklemmung die Existenz einer zyklischen Kette charakteristisch. Die Beziehungen zwischen den Philosophen und Stäbchen ist in Abbildung 3.11 graphisch verdeutlicht. Dabei steht ein Pfeil vom Stäbchen zum Philosophen dafür, dass der Philosoph das Stäbchen hat, während ein Pfeil vom Philosophen zum Stäbchen symbolisiert, dass der Philosoph das Stäbchen will.

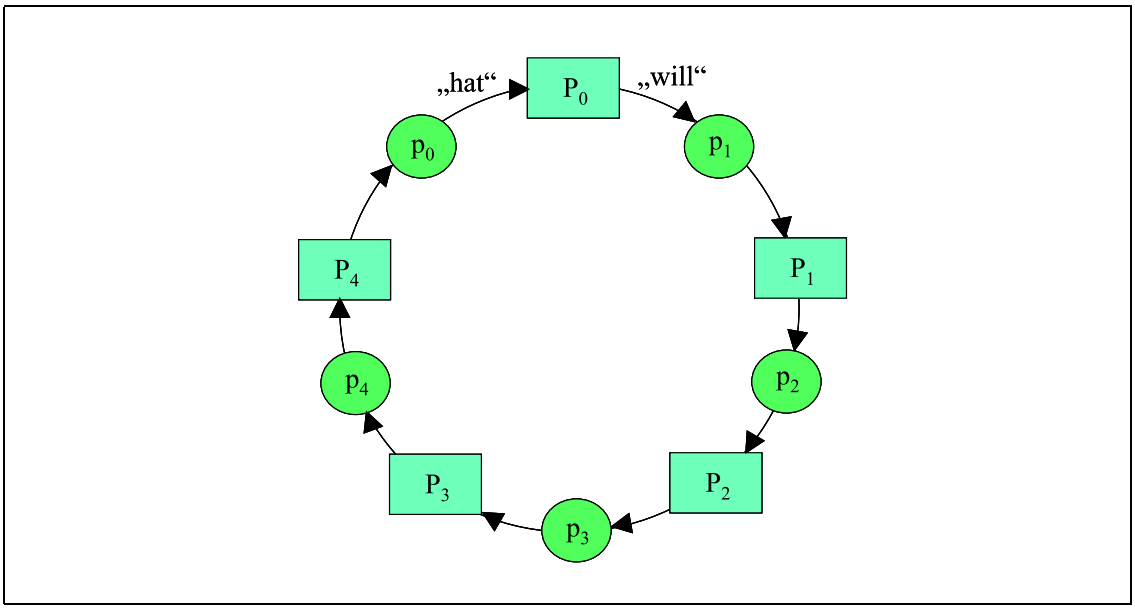


Abbildung 3.11: Zyklisches Warten der Philosophen auf die Stäbchen

Auf unseren Fall übertragen liegt ein Deadlock vor, wenn gilt:

P_0 hat s_0 und will s_1
P_1 hat s_1 und will s_2
...
P_4 hat s_4 und will s_0

Dies ist nur möglich, wenn alle Philosophen Linkshänder sind. Ist ein Rechtshänder (hier: P_4) dabei, so gehört ein Stäbchen (hier: s_4) zu keiner zyklischen Kette, und damit ist kein Deadlock mehr möglich.

In vielen Fällen, insbesondere für die Darstellung komplexerer Sachverhalte, sind die bisher vorgestellten Konzepte einschließlich der Semaphoren noch zu unhandlich, unübersichtlich und daher fehleranfällig. Aus diesem Grund greift man oft entweder auf abstraktere Notationen (wie Petrinetze) oder höhersprachliche Konstrukte (wie Regionen und Monitore) zurück, die im Folgenden dargestellt werden.

3.4 Petrinetze

Petrinetze sind ein graphisches und mathematisches Hilfsmittel, das es erlaubt, viele der bei der Informationsübertragung und -verarbeitung auftretenden Erscheinungen in einheitlicher Weise zu beschreiben. Besonders geeignet sind Petrinetze zur Modellierung von nebenläufigen, asynchronen, verteilten, parallelen, nicht-deterministischen und/oder stochastischen Systemen. Die graphische Art der Darstellung ermöglicht eine Veranschaulichung komplexer Abläufe ähnlich wie dies etwa Flussdiagramme tun. Außerdem erlauben sie die formale (mathematische) Analyse qualitativer (Lebendigkeit, Erreichbarkeit, Verklemmungen) sowie quantitativer (Verzögerung, Durchsatz) Systemeigenschaften.

Eingeführt wurden die Petrinetze von C.A. Petri 1962 im Rahmen seiner Dissertation. Seither hat man, um den sehr unterschiedlichen Anforderungen und Zielsetzungen in den verschiedensten Anwendungsbereichen gerecht werden zu können, eine Reihe neuer Petrinetzkonzepte entwickelt, die teilweise deutlich von der ursprünglichen Idee abweichen. Einfache Petrinetzkonzepte erlauben die Anwendung kraftvoller mathematischer Verfahren zur Analyse des modellierten Systems, während komplexere Konzepte zwar das betreffende System wesentlich detaillierter nachbilden können, dafür aber mathematisch sehr schnell unzugänglich werden. Hierin liegt auch die maßgebliche Schwäche aller Petrinetze: Mit ihrer Hilfe entwickelte Modelle tendieren schnell dahin, zu umfangreich für eine sinnvolle Analyse zu werden, obwohl das mit ihnen beschriebene System eigentlich nicht übermäßig groß erscheint. Daher ist es bei ihrer Verwendung oft notwendig, spezielle Modifikationen oder Einschränkungen vorauszusetzen, um eine geeignete Anpassung an die jeweilige Anwendung zu erreichen. Erfolgreich angewendet wurden und werden Petrinetze bislang beispielsweise in der Leistungsbewertung und auf dem Gebiet der Kommunikationsprotokolle, zunehmend auch für die Modellierung und Analyse von verteilten Software- und Datenbank-Systemen, nebenläufigen und parallelen Programmen sowie einer Fülle weiterer Gebiete.

3.4.1 Grundbegriffe

Machen wir uns zunächst einmal mit den grundsätzlichen Eigenschaften von Petrinetzen und der hierbei verwendeten Terminologie vertraut.

DEFINITION

Netz

Ein **Netz** (S, T, F) sei definiert als ein endlicher Graph $X = (\{S \cup T\}, F)$ bestehend aus:

- $S = \{s_1, s_2, \ldots, s_m\}$ einer endlichen Menge von **Stellen**
- $T = \{t_1, t_2, \ldots, t_n\}$ einer endlichen Menge von **Transitionen** und
- $F \subseteq (S \times T) \cup (T \times S)$ einer Menge von **Kanten**

In der graphischen Darstellung eines Petrinetzes erscheinen die Stellen stets als Kreise, die Transitionen als Rechtecke und die Kanten als Pfeile. Ergänzend zur Definition ist festzuhalten, dass die Mengen S und T als disjunkt vorausgesetzt werden.

Wir betrachten nun eine beliebige Stelle bzw. Transition und definieren ihren Vor- und Nachbereich wie folgt:

DEFINITION

Vor- und Nachbereich

Für ein beliebiges $x \in S \cup T$ sei

$$
\begin{array}{rcll}
{\bullet x} & := & \{y | (y,x) \in F\} & \text{Vorbereich} \\
x{\bullet} & := & \{y | (x,y) \in F\} & \text{Nachbereich} \\
{}^{-}x & := & \{(y,x) | (y,x) \in F\} & \text{Eingangskanten} \\
x^{-} & := & \{(x,y) | (x,y) \in F\} & \text{Ausgangskanten}
\end{array}
$$

Die Menge $\bullet x$ nennt man **Vorbereich** einer Stelle $x \in S$ bzw. einer Transition $x \in T$. Analog heißt $x\bullet$ **Nachbereich** einer Stelle bzw. Transition. Für eine Transition $t \in T$ heißen die Elemente der Menge $\bullet t$ Eingangsstellen und die Elemente der Menge $t\bullet$ Ausgangsstellen der Transition. Weiterhin bezeichnet man die Kanten $(s,t) \in {}^{-} t$ als Eingangskanten und die Kanten $(t,s) \in t^{-}$ als Ausgangskanten der Transition t.

Durch Angabe der Elemente von S, T und F ist ein Netz vollständig beschrieben. Zur Veranschaulichung ist in Abbildung 3.12 ein Beispiel abgebildet.

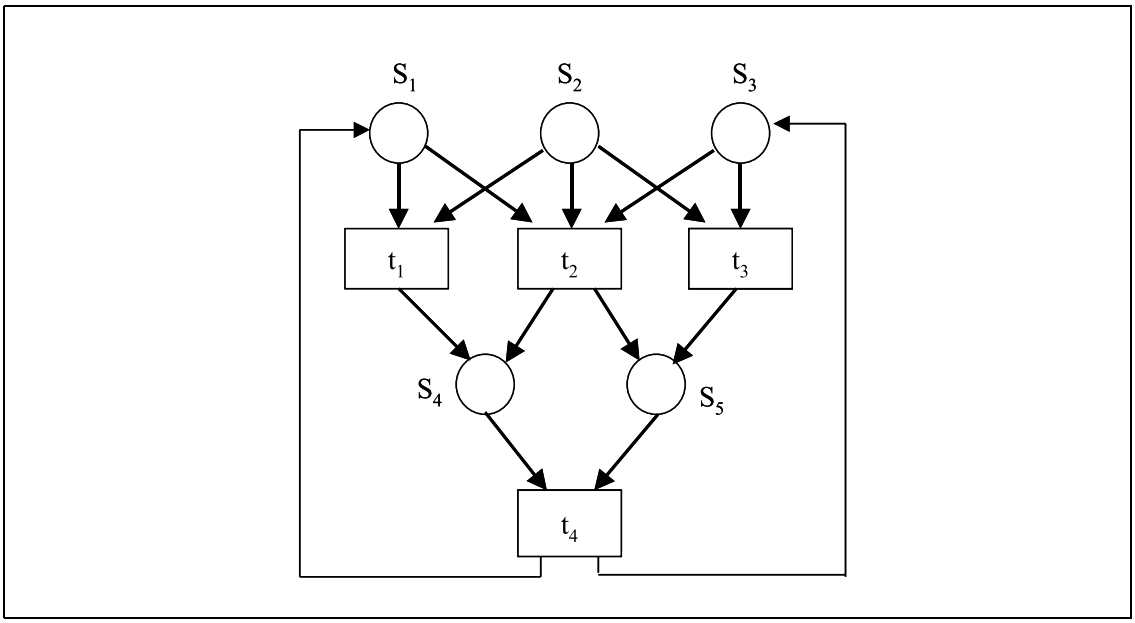


Abbildung 3.12: Petrinetz

Stellen lassen sich in einem Petrinetz je nach Anwendung meist als Zustände, bereitgestellte Daten, Signale, Bedingungen oder Speicher für Objekte interpretieren, während Transitionen üblicherweise für Ereignisse, Vorgänge, Berechnungsschritte, Jobs oder auch Prozessoren stehen.

Ein sehr gebräuchliches Petrinetz-Konzept sind Stellen-/Transitions-Systeme.

DEFINITION

S/T-System

Ein Stellen-/Transitions-System (S/T-System) ist ein 6-Tupel $Y = (S, T, F, K, W, M_0)$, bestehend aus einem Netz (S, T, F) sowie den Abbildungen

- $K : S \to \mathbb{N} \cup \{\infty\}$
- $W : F \to \mathbb{N}$
- $M : S \to \mathbb{N}$

Durch K wird jeder Stelle eine **Kapazität** zugeordnet (im einfachsten Fall beträgt diese ∞, was keine Einschränkung zur Folge hat). Die Funktion M ordnet jeder Stelle eine Anzahl so genannter Marken (Token) zu und wird daher als **Markierung** von Y bezeichnet. Dabei kann eine Stelle nicht mehr Marken aufnehmen als ihre Kapazität erlaubt, das heißt es gilt: $M(s) \leq K(s)\ \forall s \in S$. M_0 stellt die Markierung dar, wie sie zu Beginn des durch das Petrinetz modellierten Prozesses vorliegt, und heißt daher **Anfangsmarkierung**.

Die Abbildung W schließlich ordnet jeder Kante ein **Kantengewicht** zu, das im einfachsten Fall 1 beträgt. Kantengewichte von 1 werden der Übersichtlichkeit halber in der Regel nicht explizit angegeben. Umgekehrt haben Kanten, für die kein Gewicht angegeben ist, automatisch das Gewicht 1.

Die Marken werden graphisch durch schwarze Punkte innerhalb der kreisförmigen Stellen symbolisiert. Mit ihrer Hilfe kann der Gesamtzustand eines modellierten Systems dargestellt werden. Wenn die Stellen beispielsweise einen Speicher für ein spezielles Objekt darstellen, kann die Anzahl der Marken die Anzahl der vorhandenen Objekte repräsentieren. Die Kapazität einer Stelle entspricht dann der Speicherkapazität für die betreffenden Objekte.

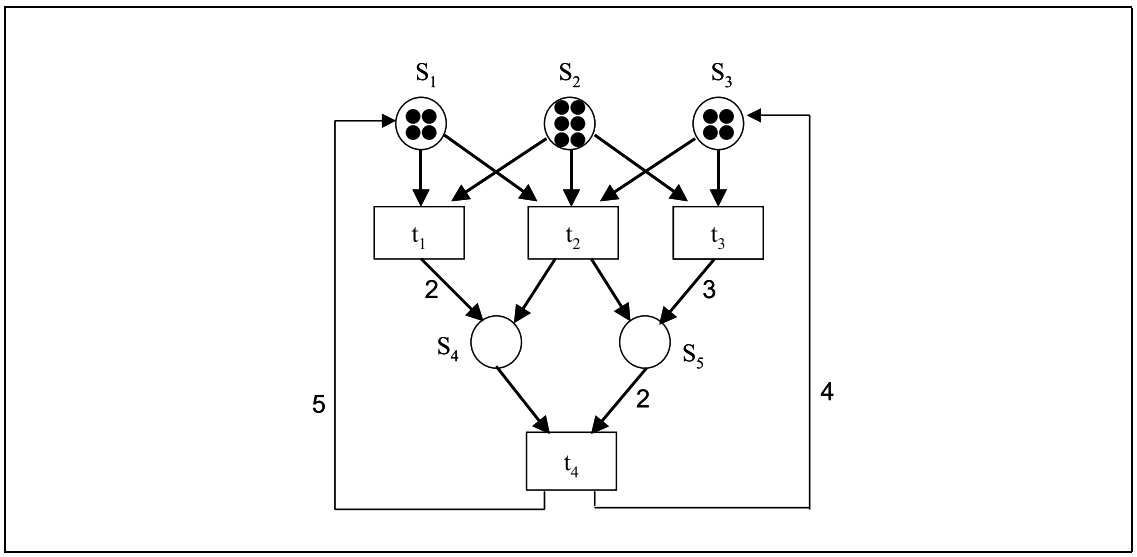


Abbildung 3.13: Petrinetz mit Token und Kantengewichten

In unserem Beispiel in Abbildung 3.13 ordnen wir jeder Stelle der Einfachheit halber die Kapazität ∞ zu, verwenden aber unterschiedliche Kantengewichte. Die gegebene Anfangsmarkierung lässt sich durch $M_0 = \{(S_1, 4), (S_2, 6), (S_3, 4), (S_4, 0), (S_5, 0)\}$ beschreiben.

Nachdem wir im ersten Schritt zunächst die Topologie des zugrundeliegenden Netzes beschrieben und als zweites dann die zur Darstellung eines Systemzustandes erforderlichen Begriffe eingeführt haben, ermöglichen wir nun das Modellieren der Dynamik eines Sy-

stems durch die Definition einer Schaltregel:

DEFINITION

Schaltregel

Man kann jeder Markierung M eine Menge $T_{akt}(M)$ zuordnen mit

$T_{akt}(M) = \{t \in T \mid (M(s) \geq W(s,t)\ \forall s \in \bullet t) \wedge (M(s) + W(t,s) \leq K(s)\ \forall s \in t\bullet)\}$

Die in $T_{akt}(M)$ enthaltenen Transitionen heißen aktiviert unter der Markierung M. Aktivierte Transitionen t können schalten (oder **feuern**). Aus der Markierung M ergibt sich durch Feuern von t die unmittelbare Folgemarkierung M' wie folgt:

$$M'(s) = \begin{cases} M(s) - W(s,t) & \text{falls } s \in \bullet t \setminus t\bullet \\ M(s) + W(t,s) & \text{falls } s \in t\bullet \setminus \bullet t \\ M(s) - W(s,t) + W(t,s) & \text{falls } s \in t\bullet \cap \bullet t \\ M(s) & \text{sonst} \end{cases}$$

Eine Transition ist also genau dann aktiviert, wenn in allen Eingangsstellen der Transition die erforderliche Anzahl von Marken vorhanden ist und die Kapazität keiner der Ausgangsstellen durch das Feuern überschritten wird. Dabei wird die Anzahl der für die Aktivierung erforderlichen Marken durch die jeweiligen Gewichte der Eingangskanten beschrieben. Beim Feuern wird die durch die jeweiligen Eingangskantengewichte festgelegte Anzahl von Marken aus den Eingangsstellen der Transition entfernt und dafür die durch die Ausgangskantengewichte beschriebene Markenzahl den Ausgangsstellen hinzugefügt. Für diesen Vorgang schreibt man auch $M[t > M'$ und meint damit, dass t unter der Markierung M zur Markierung M' schaltet. In Abbildung 3.14 ist ein Beispiel des Feuerns einer Transition dargestellt.

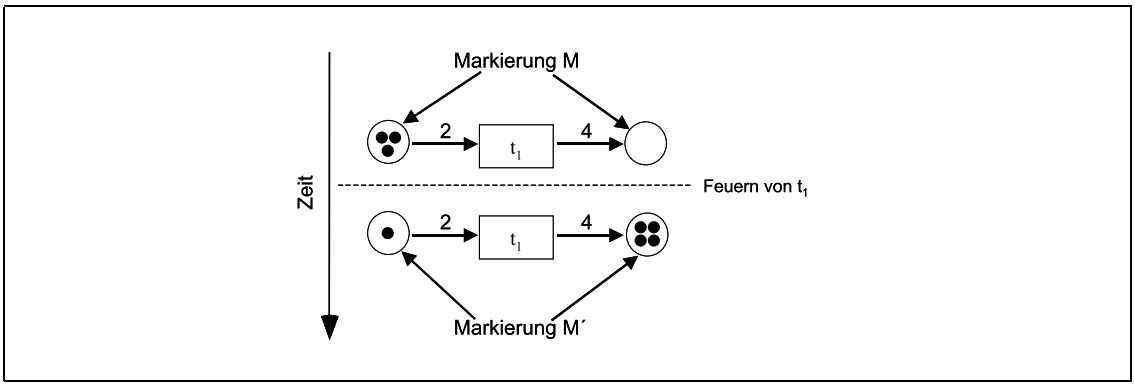


Abbildung 3.14: Übergang von einer Markierung M zu einer Markierung M'

Zur Analyse eines Petrinetzes Y gibt es zwei wichtige Hilfsmittel: die Erreichbarkeitsmenge E_Y und den Erreichbarkeitsgraphen G_Y. Es sei dazu $T = \{t_1, t_2, \ldots\}$ die Menge der einfachen Transitionen und $T^* = \bigcup_{n=0}^{\infty} T^n$. Eine Markierung M' ist von M aus erreichbar, wenn gilt:

$$\exists\, t_1 t_2 t_3 \ldots t_n \in T^*, \quad n \in \mathbb{N}_0 :\ M[t_1 > M_1[t_2 > M_2 \ldots [t_n > M'$$

Man schreibt dann auch $M[\omega M'$ mit $\omega \in T^*$. M' wird auch als mittelbare Folgemarkierung von M bezeichnet. Die **Erreichbarkeitsmenge** E_Y ist definiert durch:

$$E_Y = E_Y(M_0) := \{M' | \exists \omega \in T^* : M_0[\omega > M'\}$$

Die Erreichbarkeitsmenge enthält also die Anfangsmarkierung M_0 (da T^* auch die leere Folge enthält) und alle Markierungen des Netzes, die von der Anfangsmarkierung M_0 aus erreicht werden können.

Der Erreichbarkeitsgraph verbindet über eine Kante alle Markierungen M, $M' \in E_Y$, für die gilt: $\exists t_i \in T : M[t_i > M'$ (d.h. M' ist unmittelbare Folgemarkierung von M). Als Kantenbeschriftung wird die Transition angegeben, deren Schalten die entsprechende Änderung der Markierung des Netzes verursacht. Anhand des Erreichbarkeitsgraphen können nun verschiedene Eigenschaften des Petrinetzes - und damit des modellierten Systems - ermittelt werden. Wichtige Begriffe in diesem Zusammenhang sollen hier nur informell definiert werden. Ist eine Markierung $M \in E_Y$ nicht von jeder anderen Markierung aus erreichbar (über das Schalten mindestens einer beliebigen Folge von Transitionen), spricht man von einer teilweisen Verklemmung. Wenn es sogar für eine Markierung $M \in E_Y$ gar keine Nachfolgemarkierung M' mit $M[t > M'$ gibt, dann spricht man von einer Verklemmung oder einem Deadlock. Ist eine Transition unter keiner Markierung $M \in E_Y$ aktiviert, so spricht man von einer toten Transition. Die Transition $t_1 \in T$ steht im Konflikt mit $t_2 \in T$ unter einer Markierung M, wenn $\{t_1, t_2\} \subseteq T_{akt}(M)$, aber $t1 \notin T_{akt}(M')$ mit $M[t2 > M'$.

3.4.2 Das Erzeuger-Verbraucher-Problem

Die Anschaulichkeit, die man für eigentlich komplexe Abläufe durch die Modellierung als Petrinetz erreichen kann, soll am Beispiel des Erzeuger-Verbraucher-Problems demonstriert werden. Dieses Problem lässt sich durch das in Abbildung 3.15 gezeigte Petrinetz modellieren.

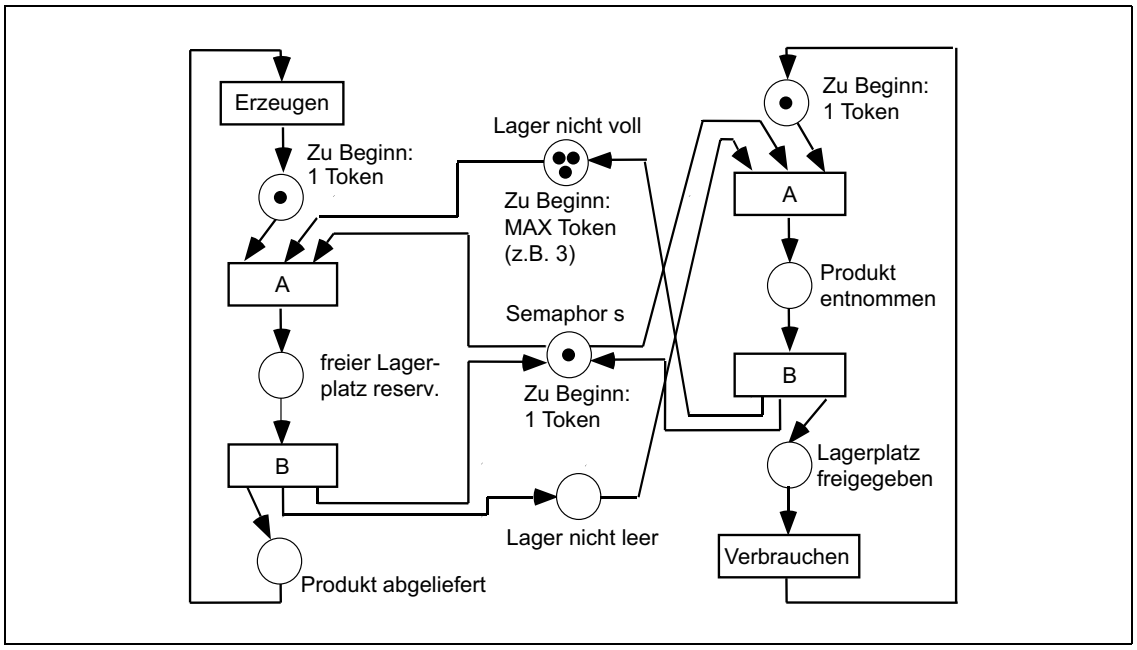


Abbildung 3.15: Das Erzeuger-Verbraucher-Problem als Petrinetz

Zu Beginn liegen ein Token in s, MAX Token in empty und kein Token in full. Was dann passiert, mag der geneigte Leser am besten selbst (unter Zuhilfenahme diverser

Pfennigstücke, Reissnägel o. ä.) ausprobieren. Sehr schnell und sehr schön wird dabei deutlich werden, dass die Dynamik dieses Petrinetzes genau der Lösung des Erzeuger-Verbraucher-Problems entspricht, wie sie unter in Abschnitt 3.3.2 algorithmisch dargestellt wurde.

3.5 Bedingte kritische Regionen

Ein signifikanter Nachteil der Semaphoren besteht darin, dass sie gerade bei der Lösung komplexer Aufgaben unübersichtlich und daher fehleranfällig sind. Insbesondere kann ein versehentliches Vertauschen von signal- und wait-Befehlen schnell fatale Folgen haben. Um solche Fehler auszuschließen, wurden eine Reihe von High-Level-Konstrukten eingeführt, von denen wir als erstes das Konzept der **kritischen Regionen** betrachten werden. Dabei müssen wir voraussetzen, dass sich ein Prozess zunächst einmal aus einer Reihe lokaler Daten und einem sequenziellen Programm zusammensetzt, das auf diesen Daten operiert. Der Zugriff auf diese lokalen Daten ist dabei wohlgemerkt nur dem zugehörigen sequenziellen Programm möglich. Darüber hinaus aber greifen mehrere Prozesse noch auf gemeinsame globale Daten zu.

Beim Konzept der kritischen Regionen sind solche globalen Variablen `v` vom Typ `T` folgendermaßen zu deklarieren:

```
var v: shared  T;
```

Damit wird angedeutet, dass `v` von mehreren Prozessen manipuliert werden kann. Ein Prozess kann allerdings auf die Variable `v` nur innerhalb eines Statements `region` der Form

```
region v do S;
```

zugreifen. In dieser Form ist das Konstrukt allerdings noch wenig hilfreich für Synchronisationen. Deshalb verwendet man meistens bedingte kritische Regionen:

```
region v when B do S;
```

Was bedeutet dies? Zunächst einmal wird durch `region v` sichergestellt, dass kein anderer Prozess auf die Variable `v` zugreifen kann, solange der ursprüngliche Prozess mit der Abarbeitung von `S` beschäftigt ist. Der Ausdruck `B` ist eine Boole'sche Bedingung, die den Zutritt zum kritischen Bereich einschränkt. Versucht nämlich ein Prozess, seinen kritischen Bereich zu betreten (und darin die globalen Variablen `v` zu manipulieren), wird zunächst seine Bedingung `B` ausgewertet. Lautet das Ergebnis dieser Auswertung `TRUE`, darf der betreffende Prozess in seinen kritischen Bereich, und `S` wird ausgeführt. Ist `B` aber `FALSE`, lässt der Prozess seinen kritischen Bereich in Ruhe (und legt sich beispielsweise schlafen). Später wird überprüft, ob einer der (von schlafendem Prozess zu Prozess u.U. verschiedenen) Ausdrücke `B` inzwischen `TRUE` geworden ist. Ist dies der Fall, wird einer der entsprechenden schlafenden Prozesse geweckt und kann seinen kritischen Bereich betreten. Es sei nochmals darauf hingewiesen, dass `region` die atomare Ausführung des gesamten Bereichs `S` sicherstellt. Dabei kann an der Stelle einer einzelnen Variablen `v` natürlich auch eine ganze Liste von Variablen stehen, die dann allesamt dem Zugriff anderer Prozesse entzogen bleiben, solange `S` nicht komplett ausgeführt ist.

Betrachten wir als Beispiel das Erzeuger-Verbraucher-Problem mit einem ringförmigen Zwischenlager. Dieser Puffer werde realisiert durch ein Feld `pool[0..MAX-1]`. Darüber hinaus verwenden wir noch eine Zählvariable `counter`, die uns den Lagerstand angibt (`0` $\leq$ `counter` $\leq$ `MAX`). Außerdem müssen wir noch eine Variable `buffer` definieren, in der die globalen Variablen eingekapselt sind:

```
var buffer: shared record
              pool: array[0..MAX-1] of items;
              counter, in, out: integer;
            end;
```

Dann können wir die Algorithmen folgendermaßen angeben:

PROGRAMM

Erzeuger

```
region buffer when counter < MAX do
begin
  pool[in] := nextp;
  in := (in + 1) mod MAX;
  counter := counter + 1;
end;
```

PROGRAMM

Verbraucher

```
region buffer when counter > 0 do
begin
  nextc := pool[out];
  out := (out + 1) mod MAX;
  counter := counter - 1;
end;
```

Zum Abschluss dieses Abschnitts wollen wir uns noch überlegen, wie man das Konstrukt einer bedingten kritischen Region implementieren kann. Eine mögliche Lösung greift auf Semaphore mit assoziierten Warteschlangen zurück, in welchen Prozesse unterkommen, die auf das Eintreffen von `B = TRUE` warten müssen. Dabei ist dieses Warten seinerseits noch zweistufig zu behandeln, sodass insgesamt mit jeder globalen Variablen folgende Variable assoziiert sind:

```
var
  mutex, firstdelay, seconddelay  :semaphore;
  firstcount, secondcount         :integer;
```

Die Idee dieser Implementierung ist folgende: Unter den Prozessen, die in ihren kritischen Bereich wollen, kursiert ein (durch `mutex` geschütztes) Ticket, dessen Besitz einem Prozess erlaubt, exklusiv diverse Aktivitäten (wie sie gleich näher beschrieben werden) auszuüben. Es ist also unter den eintrittswilligen Prozessen immer nur einer aktiv, alle anderen warten in einer der drei Warteschlangen `mutex` (außerhalb), `firstdelay` oder `seconddelay` (innerhalb). Bekommt ein Prozess zum ersten Mal das Ticket, überprüft er seine Bedingung `B`. Ist sie erfüllt, darf er in seinen kritischen Bereich, ist sie nicht erfüllt, reiht er sich in die erste Schlange (`firstdelay`) ein. Das Ticket wird unter gewissen Voraussetzungen (s. u.) dem nächsten Neuankömmling zur Verfügung gestellt. Auf diese Weise können sich eine

Anzahl von Prozessen mit unerfüllten Bedingungen in der ersten Schlange versammeln. Jedesmal wenn nun ein Prozess erfolgreich war und seinen kritischen Bereich wieder verlässt, geschieht folgendes: Zunächst werden alle in der ersten Schlange wartenden Prozesse am Ende der zweiten Schlange (`seconddelay`) eingereiht, sodass die erste Schlange leer wird. Dann überprüfen der Reihe nach die ältesten der Prozesse in der zweiten Schlange von neuem ihre Bedingungen. Sind diese immer noch nicht erfüllt, reihen sich die Prozesse wieder in die erste Schlange ein. Sobald sich bei dieser Überprüfung ein Prozess mit `B = TRUE` findet, darf dieser in seinen kritischen Bereich, arbeitet sein `S` ab, verlässt seinen kritischen Bereich und veranlasst wieder, dass alle Prozesse in der ersten Schlange sich zur zweiten begeben und daraufhin alle Prozesse der zweiten Schlange wieder ihre Bedingungen überprüfen usw. Findet sich bei einer solchen Überprüfung kein Prozess mit `B = TRUE`, wird das Ticket dem nächsten Neuankömmling zur Verfügung gestellt.

Die Umsetzung dieser Idee in einen Algorithmus könnte wie folgt aussehen:

PROGRAMM

Implementierung von bedingten kritischen Regionen

```
wait(mutex);
while not B do
begin
  firstcount := firstcount + 1;
  if secondcount > 0 then
    signal(seconddelay)
  else
    signal(mutex);
  wait(firstdelay);
  firstcount := firstcount - 1;
  secondcount := secondcount + 1;
  if firstcount > 0 then
    signal(firstdelay)
  else
    signal(seconddelay);
  wait(seconddelay);
  secondcount := secondcount - 1;
end;

Kritischer Bereich;

if firstcount > 0 then
  signal(firstdelay)
else if secondcount > 0 then
  signal(seconddelay)
else
  signal(mutex);
```

Dabei warten Neuankömmlinge in der mit `mutex` assoziierten Schlange, bis sie erstmals aufgerufen werden. Die erste Schlange entspricht dem Semaphor `firstdelay`, analog die zweite Schlange dem Semaphor `seconddelay`. Die Anzahl der in diesen beiden Schlangen wartenden Prozesse wird von `firstcount` bzw. `secondcount` aufgezeichnet.

BEISPIEL

Bedingte kritische Region

Zum Abschluss betrachten wir ein Beispiel zu diesem Algorithmus. Nehmen wir an, fünf Prozesse $1, 2, \ldots, 5$ scheitern jeweils an ihrem B.

firstdelay:	1 2 3 4 5
seconddelay:	

Dann verlässt ein weiterer Prozess seinen kritischen Bereich. Auf sein `signal` hin wandern alle Prozesse in die zweite Schlange:

firstdelay:	
seconddelay:	1 2 3 4 5

und überprüfen dann der Reihe nach ihre Bedingungen. Prozess 3 findet als erster `B = TRUE`, betritt seinen kritischen Bereich

firstdelay:	1 2
seconddelay:	4 5

verlässt ihn und signalisiert. Daraufhin wandern wieder alle in der ersten Schlange wartenden Prozesse in die zweite

firstdelay:	
seconddelay:	4 5 1 2

und das Spiel beginnt von vorne.

3.6 Monitore

Ein weiteres High-Level-Konstrukt zur Koordination bzw. Synchronisation von Prozessen begegnet uns in Gestalt des Monitorkonzepts. Ein **Monitor** ist dabei ein Objekt, das sich im Wesentlichen aus einer Menge von Prozeduren und Daten zusammensetzt. Entscheidend ist, dass zu jeder Zeit der Monitor (bzw. eine der in ihm enthaltenen Prozeduren) stets nur von höchstens einem Prozess genutzt werden darf. Ruft also ein Prozess eine Monitorprozedur auf und erhält die Erlaubnis, sie abzuarbeiten, läuft die Prozedur atomar ab. Man könnte einen Monitor beispielsweise mit einem Formel-I-Rennwagen vergleichen. Auch hier handelt es sich um ein High-Level-Konstrukt, das seinen Benutzerprozessen (d.h. den Herren Schumacher, Berger, Hill usw.) eine größere Anzahl u.U. ausgefeilter Funktionalitäten (vorwärtsfahren, funken, Krach machen etc.) zur Verfügung stellt, aber stets nur von höchstens einem Prozess benutzt werden kann.

Die Syntax eines Monitors hat i. d. R. folgende Gestalt:

PROGRAMM

Monitor

```
monitor name
begin
  Variablendeklarationen;
  procedure proc1;
  procedure proc2;
  ...
  Vorbesetzung der Variablen;
end;
```

Da sich immer nur höchstens ein Prozess innerhalb des Monitors aufhalten darf, braucht sich der Programmierer keine Gedanken um die explizite Implementierung von Sperren vor einem kritischen Bereich zu machen. Eine schematische Sicht eines Monitors ist in Abbildung 3.16 abgebildet.

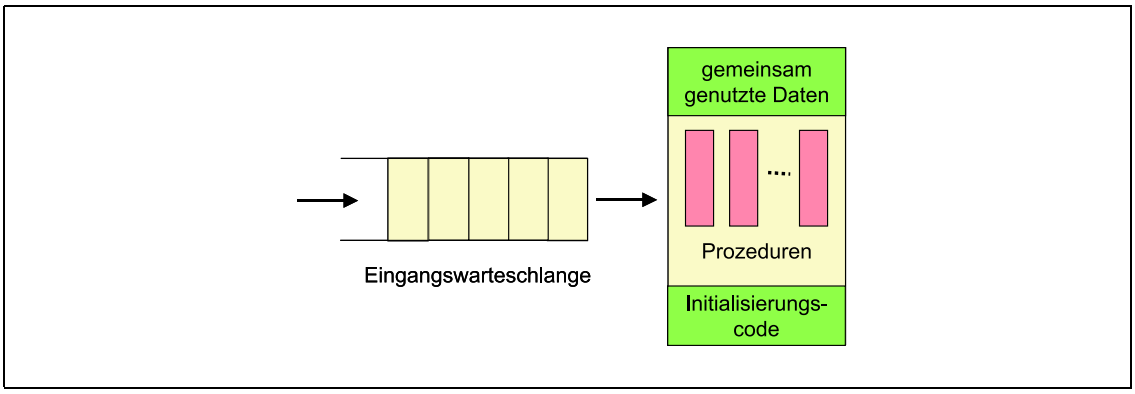


Abbildung 3.16: Schematische Sicht eines Monitors

Indes sind Monitore in dieser Gestalt noch nicht zur Synchronisation von Prozessen geeignet. Es fehlt noch (ähnlich wie bei den bedingten kritischen Regionen) eine Möglichkeit, die Abarbeitung der Monitorprozeduren vom Eintreten gewisser Bedingungen abhängig zu machen. Hierzu dient das Konzept der **Bedingungsvariablen** (vom Typ `condition`). Diese Variablen werden ähnlich gehandhabt wie Semaphore, insbesondere gibt es wie bei diesen Wait- und Signal-Operationen.

Sei also beispielsweise `cond` eine Variable vom Typ `condition`. Der Befehl `wait(cond)` in einer Monitorprozedur (eine andere mögliche Syntax wäre `cond.wait`) veranlasst einen Prozess, sich schlafenzulegen (also sich hinten in die zu `cond` assoziierte Warteschlange einzureihen). `signal(cond)` (bzw. `cond.signal`) weckt analog dazu den ältesten Prozess der zugehörigen Schlange. Im Unterschied zum Semaphorkonzept hat ein Signal jedoch keine Wirkung, wenn im Augenblick des Absendens kein anderer Prozess darauf wartet. Man tut in diesem Fall einfach so, als sei kein Signal gegeben worden (der aufmerksame Leser wird deswegen auch bei allen folgenden Beispielen feststellen, dass jedem `wait`-Befehl immer noch eine Abfrage vorgeschaltet wird, ob ein solches verpufftes Signal gesendet wurde). Man kann dies auch dahingehend interpretieren, dass `signal(cond)` das Eintreffen einer Bedingung `cond` signalisiert, während `wait(cond)` das Warten von Prozessen aufgrund des Nichterfülltseins von `cond` realisiert.

Eine Synchronisation mittels `wait` und `signal` schafft natürlich gewisse Komplikationen, da zu jedem Zeitpunkt nur ein einziger Prozess den Monitor nutzen darf. Was passiert aber, wenn ein solcher Prozess P_0 während seiner Monitorbenutzung ein `signal` von sich gibt, auf das ein weiterer Prozess P_1 wartet, um seinerseits fortschreiten zu können? Of-

fensichtlich muss sich entweder P_0 sofort schlafen legen oder P_1 darf nicht geweckt werden, denn nur so lässt sich gewährleisten, dass keine zwei Prozesse gleichzeitig aktiv sind.

Auf unser Rennwagenbeispiel lässt sich dies folgendermaßen übertragen: Zum Testen eines neuen Boliden werden zwei Fahrer (P_0 und P_1) herangezogen, die vertragsgemäß beide zehn Runden zu absolvieren haben. Am Morgen kommt einer der beiden (P_0) an und fährt los. Kurze Zeit später erscheint auch der andere Fahrer (P_1), stellt sich an den Straßenrand und wartet. In der siebten Runde bekommt Fahrer P_0 plötzlich rasende Kopfschmerzen und signalisiert der Box, dass er keine Lust mehr hat, weiterzufahren. Für solche Situationen können zwei verschiedene Vorgehensweisen (= Semantiken) festgelegt worden sein: Entweder unterbricht Fahrer P_0 seine Fahrt, Fahrer P_1 besteigt den Wagen (= Monitor) und fährt seine zehn Runden, steigt wieder aus, und Fahrer P_0 fährt danach noch vertragsgemäß seine drei Runden zu Ende. Oder aber Fahrer P_0 zeigt sich mannhaft, nimmt eine Aspirin und seine Kopfschmerzen nicht zur Kenntnis und fährt seine zehn Runden zu Ende, bevor er das Auto verlässt und Fahrer P_1 weiterfahren kann.

Eine Lösung dieser Problematik besteht darin, den Befehl `signal(cond)` nur als die allerletzte Operation einer Monitorprozedur zu gestatten (da der fragliche Prozess P_0 daraufhin den Monitor sowieso verlässt, gibt es kein Problem). Dies hätte zur Folge, dass in der Formel 1 Kopfschmerzen per Definition nur am Ende der 10. Runde auftreten können. Da dies allerdings nicht immer möglich ist (z.B. wenn mehr als ein `signal` zu geben ist), bleibt nichts anderes übrig als eine ausdrückliche semantische Festlegung zu treffen. Wenn Prozess P_0 `signal(cond)` gibt und Prozess P_1 auf `cond` wartet, muss man sich zwischen folgenden möglichen Semantiken entscheiden: Entweder legt sich P_0 schlafen und wartet solange, bis P_1 den Monitor wieder verlassen hat, oder P_0 fährt mit der Monitorbenutzung fort, wobei P_1 weiterhin wartet. Beide Möglichkeiten haben ihre Vorzüge. Da sich P_0 bereits im Monitor befindet, scheint der zweite Vorschlag vernünftiger zu sein, hat aber den Nachteil, dass zu dem Zeitpunkt, zu dem P_1 schließlich Zutritt zum Monitor erhält, die ursprünglich abgewartete Bedingung möglicherweise bereits wieder veraltet ist.

3.6.1 Exklusive Vergabe eines Betriebsmittels

Im Folgenden wollen wir anhand einiger (z. T. bereits besprochener) Beispiele die Nutzung von Monitoren demonstrieren.

Beginnen wir mit dem Monitor EXKL, der den exklusiven Zugriff auf ein gemeinsam genutztes Betriebsmittel regeln soll. Der verwendete Monitor stellt hierzu die Prozeduren `belegen` und `freigeben` bereit, sodass die exklusive Nutzung des Betriebsmittels durch das folgende, sehr übersichtliche Programmstück gewährleistet werden kann:

PROGRAMM

Verwendung von Monitoren

```
EXKL.belegen;
Kritischer Bereich;
EXKL.freigeben;
```

Zur Implementierung des Monitors werden die Variable `frei` vom Typ `condition` und die Boole'sche Variable `busy` verwendet. Bevor ein Prozess das Betriebsmittel belegen kann, muss überprüft werden, ob es bereits belegt ist. Ist dies der Fall, legt sich der Prozess solange schlafen, bis ein `signal(frei)` aufgerufen wurde. Nachdem der schlafende Prozess aufgeweckt wurde, zeigt er durch `busy := TRUE` an, dass das Betriebsmittel bereits benutzt wird. Nachdem die Betriebsmittelnutzung beendet wurde, wird `busy := FALSE`

gesetzt und durch `signal(frei)` einer der evtl. schlafenden Prozesse aufgeweckt. Eine Implementierung des Monitors ist im Folgenden angeben:

PROGRAMM

Monitor für ein exklusives Betriebsmittel

```
monitor EXKL;
begin

var busy: boolean;
var frei: condition;

procedure belegen;
begin
  if busy then
    wait(frei);
  busy := TRUE;
end;

procedure freigeben;
begin
  busy := FALSE;
  signal(frei);
end;

busy := FALSE;       (* Vorbesetzung *)
end;
```

3.6.2 Das Erzeuger-Verbraucher-Problem

Als zweites Beispiel betrachten wir einen Monitor namens `LAGER` zur Lösung des Erzeuger-Verbraucher-Problems. Dieser Monitor stellt den Erzeugern eine Prozedur `Einfügen(x)` bereit, mit Hilfe derer sie ein erzeugtes Produkt `x` in das Lager legen können. Umgekehrt können Verbraucher durch Aufruf der Prozedur `Entnehmen(x)` das nächste Produkt aus dem Lager nehmen und dieses verarbeiten. Die Benutzung des Monitors durch die Erzeuger und Verbraucher ist auch hier sehr übersichtlich:

PROGRAMM

Erzeuger

```
repeat
  erzeuge x;
  LAGER.Einfügen(x);
until FALSE;
```

PROGRAMM

Verbraucher

```
repeat
  LAGER.Entnehmen(x);
  verbrauche x;
until FALSE;
```

Wir setzen voraus, dass der gemeinsame Puffer die Kapazität MAX besitzt. Ferner verwenden wir die Zählvariablen `lastpointer` zur Anzeige des nächsten zu nutzenden Lager-

platzes und `count` zum Zählen der derzeit im Lager befindlichen Produkte. Des Weiteren werden die Bedingungsvariablen `nonempty` und `nonfull` verwendet, um einerseits das Vorhandensein eines Produktes und andererseits das Vorhandensein eines freien Platzes zu signalisieren.

PROGRAMM

Lösung des Erzeuger-Verbraucher-Problems mit Monitoren

```
monitor LAGER;
begin

var buffer: array[0..MAX-1] of items;
var lastpointer, count: integer;
var nonempty, nonfull: condition;

procedure Einfügen(x);
begin
  if count = MAX then
    wait(nonfull);
  lastpointer := lastpointer + 1 mod MAX;
  count := count + 1;
  buffer[lastpointer] := x;
  signal(nonempty);
end;

procedure Entnehmen(x);
begin
  if count = 0 then
    wait(nonempty);
  x := buffer[(lastpointer - count + 1) mod MAX]; (* älteste Einheit! *)
  count := count - 1;
  signal(nonfull);
end;

count := 0;
lastpointer := 0;     (* Vorbesetzung *)
end;
```

3.6.3 Das dritte Reader-Writer-Problem

Als letztes Beispiel für einen Monitor betrachten wir die faire dritte Variante des **Reader-Writer-Problems**: Ein ankommender Reader erhält Priorität vor später ankommenden Writern. Verlässt ein Writer den kritischen Bereich, erhalten evtl. wartende Reader Zutritt, bevor der nächste Writer Einlass erhält. Der Monitor `RW` stellt hierzu die vier Prozeduren `startread`, `endread`, `startwrite` und `endwrite` bereit. Um die derzeit aktiven Reader zu zählen, verwenden wir die Zählvariable `readercount`. Die Boole'sche Variable `busywrite` zeigt an, ob zur Zeit ein Writer aktiv ist. Des Weiteren werden die Bedingungsvariablen `okread` und `okwrite` verwendet, um das Warten eintreffender Reader bzw. Writer zu koordinieren. Für den Fall, dass ein Reader einen aktiven oder wartenden Writer vorfindet, legt sich der Reader mittels `wait(okread)` solange schlafen, bis ein `signal(okread)` gegeben wird. Analog dazu legt sich ein Writer mittels `wait(okwrite)` schlafen, wenn er entweder einen aktiven Writer oder mindestens einen aktiven Reader vorfindet. Der Zustand der beiden Warteschlangen kann mittels `okread.nonempty` bzw. `okwrite.nonempty` abgefragt werden.

PROGRAMM

Lösung für 3. R/W mit Monitoren

```
monitor RW;

begin

var readercount: integer;
var busywrite: boolean;
var okread, okwrite: condition;

procedure startread;
begin
  if (busywrite or okwrite.nonempty) then
    wait(okread);
  readercount := readercount + 1;
  signal(okread);
end;

procedure endread;
begin
  readercount := readercount - 1;
  if readercount = 0 then
    signal(okwrite);
end;

procedure startwrite;
begin
  if (busywrite or (readercount > 0)) then
    wait(okwrite);
  busywrite := TRUE;
end;

procedure endwrite;
begin
  busywrite := FALSE;
  if okread.nonempty then
    signal(okread)
  else
    signal(okwrite);
  end;

  readercount := 0; busywrite := FALSE;
end;
```

Auch hier noch die zugehörigen Routinen der Reader und Writer:

PROGRAMM

Reader

```
repeat
  RW.startread;
  Lesen;
  RW.endread;
  andere Operationen;
until FALSE;
```

PROGRAMM

Writer

```
repeat
  RW.startwrite
  Schreiben;
  RW.endwrite;
  andere Operationen;
until FALSE;
```

Zur Veranschaulichung zeigen wir hier anhand des Beispiels in Abbildung 3.17, wie bei Verwendung dieser Algorithmen ein typischer Ablauf aussehen kann.

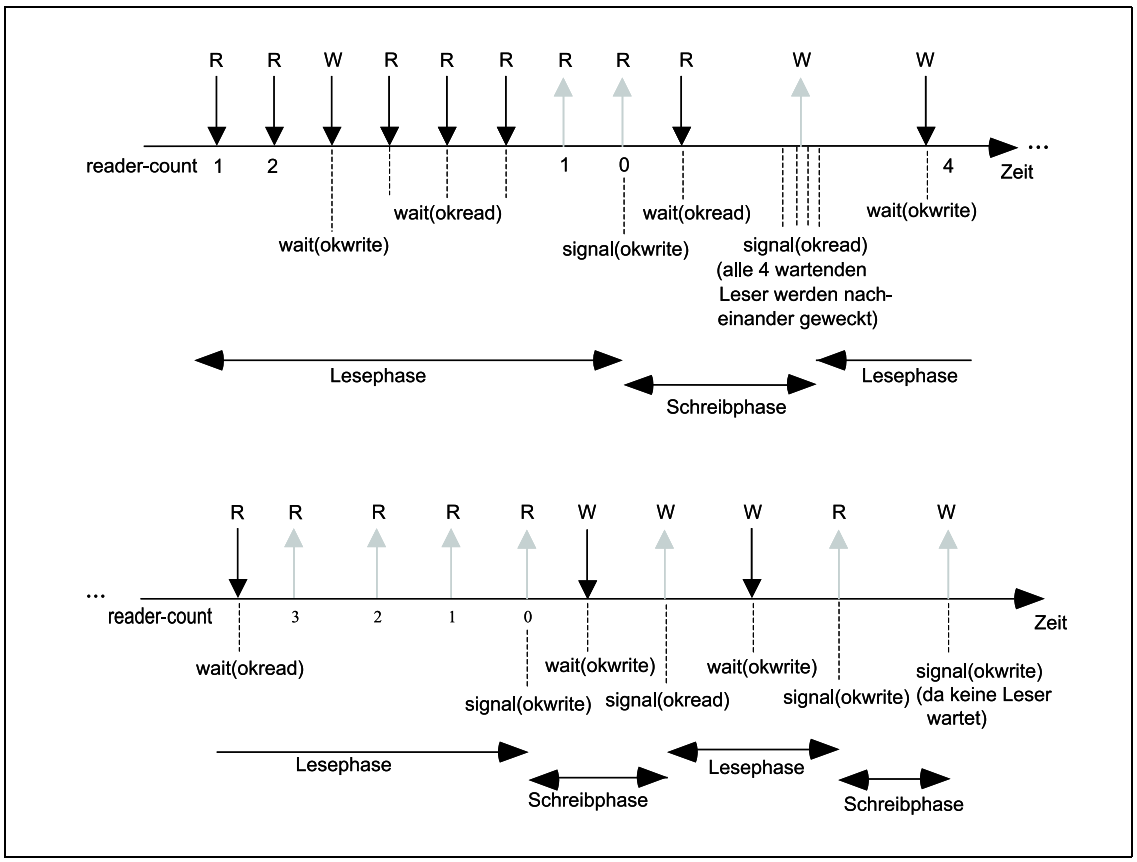


Abbildung 3.17: Ablaufbeispiel für das 3. Reader/Writer-Problem

KAPITEL 4

Datenbankorganisation

4.1 Transaktionen

Wir haben bisher eine Vielzahl von Möglichkeiten kennengelernt, die **Atomarität** von Zugriffen auf kritische Bereiche sicherzustellen. Als ein Gebiet, auf dem derartige Mechanismen auch wirklich zum Einsatz kommen, greifen wir die Organisation von Datenbanken heraus.

Die Hauptaufgaben einer Datenbank bestehen in der Sicherung, Wiedergewinnung und Konsistenzerhaltung großer Datenbestände. Dabei bedient man sich des Konzepts so genannter **Transaktionen**. Hierunter verstehen wir eine Menge von Operationen, die Lese- und Schreibzugriffe auf Datenbestände durchführen.

Als Beispiel für eine Transaktion soll die Buchung eines Fluges im Reisebüro dienen. Eine solche Buchung setzt sich zusammen aus einer Vielzahl von Zugriffen auf Informationen einer zentralen Datenbank, beispielsweise aus

- Anfrage Flugnummer
- Anfrage Platzzahl in verschiedenen Sitz- oder Preisklassen
- mögliche Tarifermäßigungen
- Rückfragen
- ....
- Buchung

Ein wichtiges Anliegen muss nun sein, die Atomarität einer solchen (Buchungs-)Transaktion zu gewährleisten. Andernfalls wäre es leicht möglich, dass beispielsweise die Anfragen zweier verschiedener Reisebüros sich ineinander verschachteln. Dies kann bei ungünstigen Konstellationen dazu führen, dass das eine Reisebüro herausfindet, dass ein bestimmter Flug noch frei ist, mit dem Buchen dann aber solange braucht, dass ein anderes Reisebüro in der Zwischenzeit die letzten freien Plätze dieses Fluges belegt, ohne dass das erste Reisebüro dies merkt. Die Konsistenz der Daten wäre somit nicht gewährleistet.

4.2 Einfache Recovery-Methoden

Transaktionen bestehen allgemein aus einer Reihe von Schreib- und Lese-Operationen. Wenn die Transaktion korrekt beendet werden konnte, wird sie durch eine **Commit**-Operation abgeschlossen. Andernfalls, d.h. beim Auftreten eines Fehlers, wird sie durch eine **Abort**-Operation abgeschlossen. Tritt ein Abort auf, muss ein sog. **Rollback** durchgeführt werden, um einen konsistenten Zustand der Daten wiederherzustellen. Eine derartige Aktion bezeichnet man auch als **Recovery-Operation**.

Gewöhnlich verwendet jedes Rechnersystem verschiedene Speichertypen. Insbesondere ist hier zu unterscheiden zwischen

- flüchtigen Speichern (*volatile storage*): Arbeitsspeicher
- nichtflüchtigen Speichern (*nonvolatile storage*): Hintergrundspeicher, Platte, Band
- stabilen Speichern (*stable storage*): CD-ROM.

Während in flüchtigen Speichern liegende Informationen leicht (z.B. bei Stromausfall) verlorengehen können, ist dies bei nichtflüchtigen Speichern kaum und bei stabilen Speichern grundsätzlich überhaupt nicht möglich. Dieser gewaltige Vorteil nichtflüchtiger Speicher wird mit einer um ein Vielfaches verlangsamten Zugriffszeit im Vergleich zu flüchtigen Speichern erkauft.

Recovery-Operationen beruhen auf dem Zusammenspiel von flüchtigen und nichtflüchtigen Speichern. Eine einfache Methode basiert dabei auf dem Führen eines **Logbuches**, das aktuelle Daten in einem nichtflüchtigen Speicher enthält. Jeder Eintrag beschreibt dabei eine einzelne Operation innerhalb einer Transaktion und enthält folgende Informationen:

- Transaktionsname,
- Namen der Daten, die neu geschrieben wurden,
- alter Wert des Datums,
- neuer Wert des Datums.

Wenn nun eine Transaktion T_i startet, wird die Eintragung „T_i `starts`“ vorgenommen. Sodann wird jede Write-Operation ins Logbuch eingetragen. Endet die Transaktion korrekt, lautet der abschließende Eintrag „T_i `commits`“.

Mit Hilfe des Logbuchs lassen sich fehlerhafte Transaktionen leicht bereinigen. Hierzu stehen zwei idempotente Prozeduren zur Verfügung:

- `undo`(T_i): durch T_i evtl. überschriebene Daten werden auf ihre alten Werte zurückgesetzt
- `redo`(T_i): alle durch T_i aktualisierten Daten werden auf ihre neuen Werte gesetzt.

Die jeweiligen alten bzw. neuen Werte lassen sich dabei dem Logbuch entnehmen. Die Idempotenz von `undo` und `redo` stellt sicher, dass es auf das Resultat keinen Einfluss hat, ob man ein und dieselbe Prozedur einmal oder mehrmals hintereinander durchführt.

Wenn nun eine Transaktion T_i nicht beendet werden konnte, kann man mittels `undo`(T_i) leicht den Zustand der Daten vor Beginn von T_i wiederherstellen. Kompliziertere Verhältnisse liegen vor, wenn ein Systemfehler aufgetreten ist. In diesem Fall ist für jede einzelne Transaktion zu entscheiden, ob sie mittels `undo` oder `redo` zu behandeln ist. Dabei verfährt man wie folgt:

- Enthält das Logbuch den Eintrag T_i `starts`, aber nicht den Eintrag T_i `commits`, so wird die Transaktion T_i mittels `undo`(T_i) komplett rückgängig gemacht.
- Enthält das Logbuch dagegen den Eintrag T_i `starts` und den Eintrag T_i `commits`, so gewinnen wir einen zuverlässigen Zustand der Daten durch die Operation `redo`(T_i).

Diese Vorgehensweise ist beim Auftreten eines Systemfehlers auf alle bislang durchgeführten Transaktionen anzuwenden. Dies kostet zum einen Zeit, zum anderen aber wird es des Öfteren vorkommen, dass durch die Operation `redo` Daten einen Wert erhalten, der im Folgenden bereits wieder veraltet ist und durch eine der nächsten Transaktionen überschrieben wird. Dadurch wird zwar die Konsistenz nicht beeinträchtigt, dennoch handelt es sich um unnötigen Overhead.

Um bei Systemfehlern die Anzahl der evtl. zu wiederholenden Operationen zu begrenzen, führt man so genannte **Checkpoints** ein. Dabei werden zu bestimmten Zeitpunkten alle derzeit in flüchtigen Speichern gehaltenen Logbuch-Einträge sowie alle in flüchtigen Speichern gehaltenen Daten auf einen stabilen Speicher übertragen. Von diesem Checkpoint an ist dann die gesamte Vergangenheit als korrekt anerkannt. Dies impliziert, dass ein Checkpoint erst dann gesetzt werden kann, wenn alle noch laufenden Transaktionen korrekt zu Ende gebracht worden sind.

Vergleichbare Vorgehensweisen findet man auch im Bereich der Datenkommunikation. Will man dort etwa eine sehr große Datei fehlerfrei übertragen, geschieht dies entweder dadurch, dass man die gesamte Datei auf einmal überträgt (und beim Auftreten eines Fehlers von vorne anfängt) oder indem man die Datei durch Checkpoints in Kapitel aufteilt (und bei einem eventuellen Fehler nur bis zum letzten korrekt verbuchten Checkpoint zurücksetzt).

Konkret geschieht dies bei Datenbanken auf folgende Art und Weise: Jeder Checkpoint wird ins Logbuch eingetragen. Tritt ein Systemfehler auf, wird die erste Transaktion nach dem letzten Checkpoint ausfindig gemacht; genauer gesagt die erste Transaktion, deren Start nach dem letzten Checkpoint ins Logbuch eingetragen wurde. Diese und alle später begonnenen Transaktionen werden dann daraufhin überprüft, ob sie jeweils einen Eintrag T_i `commits` im Logbuch stehen haben. In diesem Fall wird ein `redo`(T_i) durchgeführt, anderenfalls ein `undo`(T_i).

4.3 Serialisierbarkeit und potenzielle Konflikte

Gegeben seien Transaktionen $T_1, T_2, \ldots, T_n$, die ineinander verzahnt ausgeführt werden können. Man bezeichnet eine solche Ausführungsreihenfolge als **Schedule**. In einem Schedule können also z.B. zunächst einige Operationen der Transaktion T_1 ausgeführt werden, dann einige, die zu T_3 gehören, dann wieder einige von T_1, danach welche von T_2 usw. Ein Schedule heißt seriell, wenn er der Ausführung einer beliebigen Permutation der Transaktionen $T_1, T_2, \ldots, T_n$ entspricht, d.h. die Transaktionen werden nicht ineinander verzahnt, sondern jede einzelne auf einmal ausgeführt, dafür aber in einer beliebigen Reihenfolge.

Betrachten wir als nächstes ein ganz einfaches Beispiel für ein solches **serielles Schedule**:

BEISPIEL

Einfaches serielles Schedule

T_0	T_1
`read`(A)	
`write`(A)	
`read`(B)	
`write`(B)	
	`read`(A)
	`write`(A)
	`read`(B)
	`write`(B)

Nicht-serielle Schedules sind nun nicht notwendigerweise inkorrekt, selbst dann nicht, wenn sie Lese- und Schreiboperationen auf gemeinsamen Datenbeständen ausführen:

BEISPIEL

Korrektes nicht-serielles Schedule

T_0	T_1
`read`(A)	
`write`(A)	
	`read`(A)
	`write`(A)
`read`(B)	
`write`(B)	
	`read`(B)
	`write`(B)

Bezeichne $WM(i)$ die **Write-Menge** der Transaktion T_i, d.h. die Menge der Daten, auf die im Rahmen von T_i schreibend zugegriffen wird. Analog dazu sei $RM(i)$ die **Read-Menge** von T_i. Für die Transaktionen T_i und T_j liegt ein potenzieller Konflikt vor, falls

$$((WM(i) \cap WM(j)) \cup (WM(i) \cap RM(j)) \cup (RM(i) \cap WM(j)) \neq \varnothing$$

gilt. Ist nämlich $WM(i) \cap WM(j) \neq \varnothing$, kommt es darauf an, welche der beiden Transaktionen zuletzt schrieb; die write-Operation der anderen Transaktion geht verloren:

BEISPIEL

Verlorengehende Schreib-Operation

T_i	T_j
`write`(C)	
(geht verloren !)	`write`(C)

Ist $WM(i) \cap RM(j) \neq \varnothing$ bzw. $RM(i) \cap WM(j) \neq \varnothing$, können in einer Transaktion unmittelbar hintereinander ausgeführte Leseoperationen unterschiedliche Ergebnisse liefern:

BEISPIEL

T_0	T_1
	`read`(D) (alter Wert)
`write`(D)	
	`read`(D) (neuer Wert)

In diesem Fall ist nicht klar, welcher Wert von D der richtige ist.

Betrachten wir das obige Beispiel des korrekten nicht-seriellen Schedules, so stellen wir fest, dass auch hier beispielsweise $RM(T_0) \cap WM(T_1) \neq \varnothing$ gilt. Dies zeigt, dass in Einzelfällen die Konsistenz erhalten bleiben kann, auch wenn streng genommen ein Konfliktrisiko besteht.

Manchmal kann man durch **Swapping** (d.h. sukzessives Vertauschen) von zeitlich benachbarten und nicht konfligierenden Daten (hier: `read`(B), `write`(B) von T_0 mit `read`(A), `write`(A) von T_1) das in Konflikt stehende Schedule in ein serielles umgewandelt werden. Derartige Schedules heißen **konfliktserialisierbar**. In Abbildung Abbildung 4.1 ist die Serialisierbarkeit der beiden Transaktionen verdeutlicht. Die angedeuteten Pfeile erfordern mehrere einzelne Swap-Operationen.

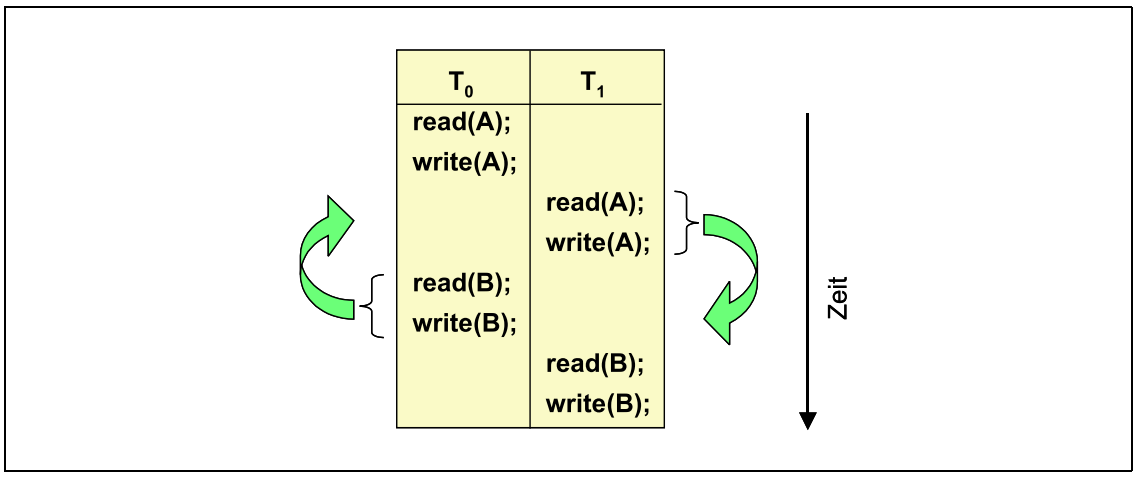


Abbildung 4.1: Konfliktserialisierbarer Schedule

Die Atomarität der einzelnen Transaktionen lässt sich dabei mittels einer wait/signal-Konstruktion leicht gewährleisten. Für allgemeine Zwecke ist dies aber oft zu umständlich. Daher lernen wir im Folgenden zwei effizientere Protokolle zur Gewährleistung der Serialisierbarkeit kennen.

4.4 Das Zwei-Phasen-Sperrprotokoll

Sperrprotokolle beruhen darauf, dass vor dem Zugriff auf Daten, die auch von anderen genutzt werden können, atomare Sperren gesetzt werden (lock), die nach dem Zugriff wieder freigegeben werden (unlock). Das Design solcher Protokolle orientiert sich vor allem an der Granularität der Sperren, also an der Frage, ob man eine geringe Anzahl von Sperren, die größere *Gebiete* innerhalb der Datenbank abschließen und daher evtl. Zugriffsbegrenzungen zur Folge haben, oder eine hohe Anzahl von Sperren samt dem zugehörigen Aufwand für deren Verwaltung vorzieht.

Ein häufig verwendetes Sperrprotokoll soll hier näher betrachtet werden, und zwar das sog. **Zwei-Phasen-Sperrprotokoll** (**Two-Phase-Locking**). Der Name rührt daher, dass sich das Vorgehen dieses Protokolls in eine Up- und eine Down-Phase unterteilen lässt. Grundsätzlich gilt dabei, dass eine Transaktion erst dann Sperren anfordert, wenn sie wirklich benötigt werden. Die erste Phase (Up) ist dabei charakterisiert durch eine zunehmende Anzahl von Sperren. Neue Sperren können nur in der ersten Phase angefordert werden. In der zweiten Phase (Down) werden dann alle aufgebauten Sperren (mehr oder

weniger) gleichzeitig wieder freigegeben (siehe Abbildung 4.2).

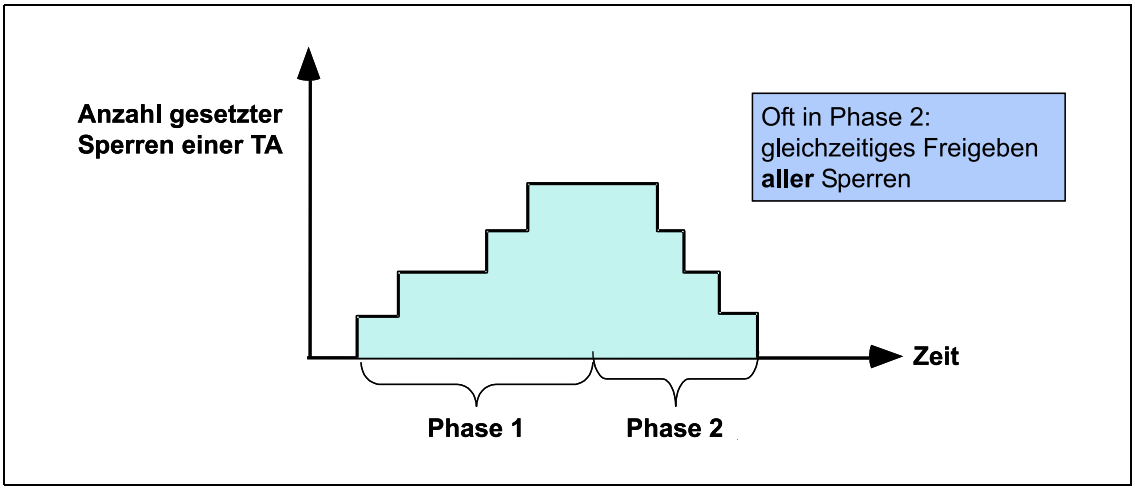


Abbildung 4.2: Prinzip des Zwei-Phasen-Sperrens

Hierbei ist also ausgeschlossen, dass beispielsweise zunächst Sperren angefordert werden, dann einige davon wieder freigegeben werden und daraufhin neue Sperren aufgebaut werden usw.

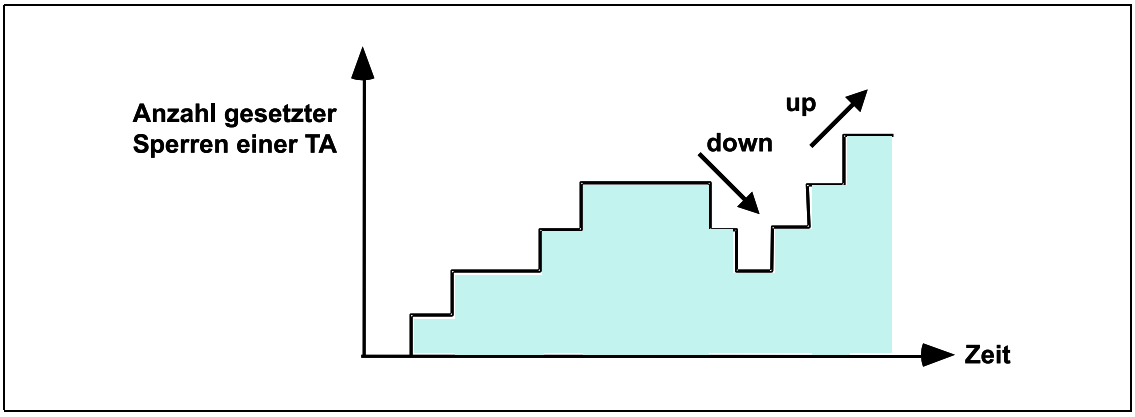


Abbildung 4.3: Unzulässige Abfolge von Up- und Down-Phasen

Warum ist es sinnvoll, ein Vorgehen wie das in Abbildung 4.3 skizzierte auszuschließen? Der Grund ist einfach: In der Down-Phase werden Datenbankeinträge verändert und anschließend freigegeben. Es ist aber möglich, dass eine Transaktion nicht korrekt beendet wird, weil sie etwa – z.B. aufgrund eines Deadlocks – zusätzlich benötigte Sperren nicht bekommt. Dann wären einige Objekte geändert, andere aber nicht, und die Konsistenz der Daten wäre nicht gewährleistet.

4.5 Timestamp-Mechanismen

Bei dem im vorherigen Abschnitt beschriebenen Sperrprotokoll wird die Reihenfolge zweier konfligierender Transaktionen durch die Zuteilung der ersten Sperre bestimmt, die beide Transaktionen gemeinsam anfordern. Daneben ist es jedoch auch denkbar, eine zeitliche Anordnung vorzunehmen. Dabei wäre jede Transaktion zu Beginn mit einem Zeitwert (**Timestamp**) zu versehen. Daraufhin hätte dann stets die älteste Transaktion (d.h. die

Transaktion mit dem kleinsten Timestamp) Vorrang. Sollten zwei Transaktionen zufällig einen identischen Zeitwert aufweisen, lässt sich eine eindeutige Entscheidung z.B. über die jeweiligen Prozessnummern treffen. Die Zeitwerte selbst kann man entweder über eine zentrale oder über lokale Uhren zuteilen. Ungenauigkeiten der lokalen Uhren führen dabei höchstens zu einer gewissen Benachteiligung von Transaktionen mit nachgehenden Uhren.

Beim Start einer Transaktion T_i wird ihr ein eindeutiger Zeitwert $TS(T_i)$ zugewiesen. Serialisierbarkeit ist dann gleichbedeutend damit, dass T_i vor T_j ausgeführt wird, sofern $TS(T_i) < TS(T_j)$ ist. Dieses Vorgehen kann noch auf Read- bzw. Write-Operationen beschränkt werden, die T_i und T_j betreffen.

Für die Implementierung eines derartigen Schemas werden jedem Datum Q zwei Zeitwerte zugeschrieben:

- $W_{TS}(Q)$ = größter Zeitwert einer Transaktion, die eine erfolgreiche Write-Operation auf Q ausgeführt hat.
- $R_{TS}(Q)$ = größter Zeitwert einer Transaktion, die Q erfolgreich gelesen hat.

Diese beiden Timestamps werden jedesmal aktualisiert, wenn eine neue `read`(Q)- oder `write`(Q)-Operation durchgeführt worden ist. Das Timestamp-Protokoll stellt sicher, dass konfligierende Lese- und Schreiboperationen in der Reihenfolge der entsprechenden Timestamps ausgeführt werden. Es operiert dabei folgendermaßen:

Sei zunächst T_i eine Transaktion, welche `read`(Q) ausführen will:

a) Falls $TS(T_i) < W_{TS}(Q)$ gilt, so heißt dies, dass eine spätere Transaktion bereits erfolgreich geschrieben hat. Q ist also bereits geändert, und es bleibt nichts anderes übrig, als `read`(Q) zu verwerfen und die Transaktion T_i neu zu starten (mit entsprechend höherem neuen Timestamp).

b) Ist $TS(T_i) \geq W_{TS}(Q)$, wird die `read`(Q)-Operation durchgeführt und anschließend der $R_{TS}(Q)$ auf das Maximum des bisherigen $R_{TS}(Q)$ und $TS(T_i)$ gesetzt.

Betrachten wir nun eine Transaktion T_i, die die Operation `write`(Q) ausführen will:

a) Ist $TS(T_i) < R_{TS}(Q)$, so wurde Q bereits früher gelesen. Würde T_i seine `write`-Operation jetzt durchführen, so hätte eine andere Transaktion einen alten Wert gelesen. Um dies zu verhindern, muss `write`(Q) verworfen werden. Die Transaktion T_i wird zurückgesetzt und neu gestartet.

b) Im Fall $TS(T_i) < W_{TS}(Q)$ würde ein Schreibvorgang ausgeführt, der bereits zum Zeitpunkt seiner Ausführung veraltet wäre. Deshalb verwirft man hier `write`(Q) und setzt die Transaktion T_i zurück.

c) Im verbleibenden Fall $TS(T_i) \geq \texttt{max}(R_{TS}(Q), W_{TS}(Q))$ gibt es keine Probleme, daher wird die Write-Operation ausgeführt.

Es sei darauf hingewiesen, dass jede zurückgesetzte Transaktion einen neuen (höheren) Timestamp erhält und mit diesem ausgerüstet nochmals gestartet wird. Das so durchgeführte Timestamp-Protokoll gewährleistet die Serialisierbarkeit konfligierender Transaktionen. Darüber hinaus bleibt festzuhalten, dass es (im Gegensatz zum Two-Phase-Locking-Protokoll) sogar Deadlockfreiheit garantiert.

4.6 Leistungsanalyse eines Sperrprotokolls

Zur Abrundung dieses Kapitels betrachten wir noch ein einfaches Modell eines Sperrprotokolls und fragen uns, was ein solches Protokoll leisten kann. Die Antwort darauf wird in einer exemplarischen Leistungsanalyse bestehen, die auf gewisse mathematische Hilfsmittel zurückgreifen wird.

Betrachten wir also das so genannte **Simple-Two-Phase-Locking-Protokoll** (Simple-2-PL), d.h. ein Zwei-Phasen-Sperrprotokoll in seiner einfachsten Variante. Hierbei sperrt eine Transaktion bei Bedarf jeweils einen Bereich der Datenbank. Bleibt die Anforderung einer solchen Sperre erfolglos, weil der betreffende Bereich bereits durch eine andere Transaktion gesperrt ist, setzen wir die Transaktion sofort zurück und geben alle von ihr gesetzten Sperren frei. Dies ist natürlich keine besonders schlaue Strategie, da es möglicherweise viel effizienter wäre, mit dem Konfliktpartner eine Einigung über den betreffenden Bereich herbeizuführen. Dieses Verhalten wird uns jedoch die Modellierung erleichtern.

Kann es hierbei zu einem Deadlock kommen? Eine Deadlocksituation könnte nur auftreten, wenn eine Transaktion T_1 eine von T_2 gehaltene Sperre anfordert, T_2 ihrerseits eine von T_1 gehaltene Sperre benötigt, und keine der beiden Transaktionen bereit ist, gehaltene Sperren zurückzugeben. Eine solche Situation ist beim Simple-2-PL offensichtlich ausgeschlossen. Es handelt sich also um ein deadlockvermeidendes, aber sehr pessimistisch konzipiertes Protokoll.

Modellbildung

Um dieses Modell mit mathematischen Methoden analysieren zu können, sind zunächst einige vereinfachende Annahmen zu treffen. Insbesondere müssen wir modellieren, wie oft Transaktionen auf unsere Datenbank zugreifen und wie deren Sperrverhalten ist, d.h. wie viele und welche Sperren für welche Zeiträume angefordert werden.

Wie kommen Anforderungen von Sperren an?

Betrachten wir eine Datenbank, die aus M gleichartigen sperrbaren Bereichen besteht. Die Ankünfte der einzelnen Transaktionen sollen möglichst zufällig (insbesondere unabhängig voneinander) sein, da wir annehmen können, dass es sich um eine sehr große Datenbank handelt, die von vielen Kunden benutzt wird, wobei letztere sich nicht gegenseitig abstimmen.

Mathematisches Standardmodell für die beschriebene Art und Weise, in der Transaktionen ankommen, ist der Poisson-Prozess. Auf ihn soll daher zunächst eingegangen werden.

Stochastische Prozesse

Ein **stochastischer Prozess** $\{X(t),\ t \in T\}$ ist eine Familie von Zufallsvariablen, d.h. für jedes t aus der Indexmenge T stellt $X(t)$ eine Zufallsvariable dar. t lässt sich dabei meist als Zeit interpretieren, und $X(t)$ ist dann nichts anderes als der Zustand des Prozesses zum Zeitpunkt t. Beispielsweise kann $X(t)$ den Saldo eines Bankkontos, die Position eines Luftmoleküls im Raum, die Anzahl von Zuhörern in einer Vorlesung oder die Anzahl von Telefonanrufen, die seit dem frühen Morgen in einem Sekretariat eingegangen sind, darstellen (und zwar jeweils zum Zeitpunkt t). Für unsere Zwecke nehmen wir an, dass die

Menge T ein Zeitintervall darstellt, der stochastische Prozess also kontinuierlich ist.

Die Menge aller möglichen Werte, welche die Zufallsvariable $X(t)$ annehmen kann, nennen wir Zustandsraum des Prozesses. Stellt $X(t)$ beispielsweise die Anzahl von Zuhörern in einer Vorlesung oder die Anzahl von Anrufen im Sekretariat dar, so ist der Zustandsraum sinnvollerweise die Menge $0, 1, 2, \ldots$ aller natürlichen Zahlen. Einen derartigen Prozess, dessen Zustandsraum die Menge der natürlichen Zahlen ist, nennt man auch **Zählprozess**.

Betrachtet man nun einen kontinuierlichen Zählprozess nur zu bestimmten Zeiten $t_0 < t_1 < \ldots < t_n$, so sind oftmals die Zuwächse $X(t_1)$ - $X(t_0)$, $X(t_2)$ - $X(t_1)$ etc. von Interesse. Man sagt, der Prozess besitze **unabhängige Inkremente**, wenn alle diese Zufallsvariablen $X(t_1)$ - $X(t_0)$, $X(t_2)$ - $X(t_1)$, ..., $X(t_n)$ - $X(t_{n-1})$ stochastisch unabhängig voneinander sind. Besitzt zusätzlich noch die Zufallsvariable $X(t_1 + s)$ - $X(t_0 + s)$ dieselbe Verteilung wie $X(t_1)$ - $X(t_0)$ für alle $t_0, t_1 \in T$ und $s > 0$, so hat der Prozess sogar *stationäre unabhängige Inkremente*.

Nehmen wir zur Veranschaulichung die Anzahl in einem Sekretariat ankommender Telefonanrufe als Beispiel für einen Zählprozess. Dieser Zählprozess hat genau dann unabhängige Inkremente, wenn die Anzahl der Anrufe, die in einem bestimmten Zeitintervall ankommen, nicht davon abhängt, wie viele Anrufe in früheren oder späteren Zeitintervallen ankommen. Die Inkremente sind genau dann stationär, wenn (unabhängig von der Tageszeit) in gleichlangen Zeitintervallen gleiches Ankunftsverhalten herrscht (insbesondere etwa gleichviele Anrufe ankommen).

Poisson-Prozesse

Mit den eingeführten Begriffen können wir nun einen Prozess charakterisieren, der das völlig zufällige Eintreten von Ereignissen in kleinen Zeitintervallen beschreibt. Betrachten wir also die Wahrscheinlichkeit dafür, dass ein Ereignis innerhalb des Zeitintervalls $[t, t+h]$ eintritt. Sinnvolle Annahmen für diese Wahrscheinlichkeit sind:

- Sie soll *proportional zur Intervall-Länge h* sein: Je länger ich warte, desto wahrscheinlicher tritt das Ereignis ein.
- Sie soll *proportional zu einer Intensitätsrate λ* sein, die die durchschnittliche Anzahl von Ereignissen pro Zeiteinheit darstellt: Je größer die Intensität, desto größer die Wahrscheinlichkeit.
- Sie soll *unabhängig von der Zeit t* sein: Es ist egal, wann ich auf das Eintreten eines Ereignisses zu warten beginne.
- Sie soll ferner *unabhängig von der Zahl und den Zeitpunkten bisheriger (und zukünftiger) Ereignisse* sein: Vor/nach mir die Sintflut.
- Außerdem soll die Anzahl, wenn man genügend kleine Zeitintervalle betrachtet, immer nur um 1 wachsen können: Die Ereignisse sollen also *selten* genug auftreten, um auszuschließen, dass zwei Ereignissen gleichzeitig (oder quasi-gleichzeitig) vorkommen.

Eine mathematische Formulierung dieser Kritierien für das völlig zufällige Eintreten von Ereignissen führt zu folgender Definition:

DEFINITION

Poisson-Prozess

Ein Zählprozess $\{N(t),\ t \geq 0\}$ heißt **Poisson-Prozess** mit Rate $\lambda > 0$, wenn gilt:

(i) $N(0) = 0$

(ii) $\{N(t),\ t \geq 0\}$ hat stationäre, unabhängige Inkremente

(iii) $P\{N(h) \geq 2\} = o(h)$

(iv) $P\{N(h) \geq 1\} = \lambda h + o(h)$

Dabei ist eine Funktion f von der Ordnung o(h), wenn gilt:

$$\lim_{h \to 0} \frac{f(h)}{h} = 0,$$

d.h. wenn $f(h)$ schneller gegen Null geht als h.

Ein derartiger Poisson-Prozess gehorcht der so genannten **Poisson-Verteilung**. Damit lässt sich die Wahrscheinlichkeit dafür, dass im Intervall $[0, t]$ genau i Ereignisse eintreten, darstellen als

$$P\{N(t) = i\} \ = \ \frac{(\lambda t)^i}{i!} \cdot e^{-\lambda t}.$$

Insbesondere ergibt sich hieraus der Mittelwert (Erwartungswert) der Ankünfte in dem betrachteten Intervall $[0, t]$ zu λt. Er ist also proportional sowohl zur Intervallänge als auch zur Intensitätsrate, wie wir es gefordert haben.

Die Poisson-Verteilung liefert übrigens oft ausgezeichnete Approximationen, wenn man Prozesse betrachtet, die aus sehr vielen voneinander unabhängigen Teilkomponenten zusammengesetzt sind.

Wie lange benutzt eine Transaktion eine neue Sperre?

Nachdem wir in den Poisson-Prozessen ein geeignetes Modell für das Ankunftsverhalten von Sperrenanforderungen gefunden haben, ist nun noch zu klären, wie lange eine Transaktion nach der Anforderung einer neuen Sperre rechnen kann, bis sie die nächste Sperre anfordern muss.

Versuchen wir wieder, ein möglichst einfaches Modell zu beschreiben. Am allereinfachsten ist es sicherlich, wenn wir fordern, dass das Eintreten des nächsten Ereignisses unabhängig von der Vergangenheit geschehen soll. Ein solches Modell nennen wir **memoryless**.

Diese Bedingung, dass die **Zwischenankunftszeit** z, also der zeitliche Abstand zwischen zwei Ereignissen, von der Vergangenheit unabhängig sein soll, wird von der (negativen) **Exponentialverteilung** erfüllt:

$$P\{z \leq t\} \ = \ 1 - e^{-\lambda t}.$$

Betrachtet man ein sehr kleines t, fragt also nach der Wahrscheinlichkeit dafür, dass der Abstand zwischen zwei Ereignissen winzig ist, so erhält man $P(z \leq t) \approx 1 - 1 = 0$. Der Fall, dass zwei Ereignisse fast gleichzeitig stattfinden, kommt also so gut wie nicht vor. Sieht man sich das ganze für sehr große t an, ergibt sich eine Wahrscheinlichkeit $P(z \leq t) \approx 1 - 0 = 1$, d.h. wenn ich genügend lange warte, passiert fast sicher irgendwann etwas. Der Parameter λ übernimmt dabei im Wesentlichen wieder die Rolle einer Intensität: mit wachsendem λ treten Ereignisse tendenziell früher ein.

Erfüllt die Exponentialverteilung tatsächlich die Memoryless-Eigenschaft? Um dies zu überprüfen, berechnen wir die Wahrscheinlichkeit dafür, dass wir auf das Eintreten des nächsten Ereignisses nochmals eine Zeit von t_2 warten müssen, nachdem wir schon eine Zeit von t_1 gewartet haben. Mit Hilfe von bedingten Wahrscheinlichkeiten ergibt sich dafür

$$P\{z \geq t_2 + t_1 \mid z \geq t_1\} \;=\; \frac{e^{-\lambda(t_2+t_1)}}{e^{-\lambda t_1}} \;=\; e^{-\lambda t_2} \;=\; P\{z \geq t_2\}.$$

Anders ausgedrückt: Wie lange ich von einem bestimmten Zeitpunkt an auf das Eintreffen des nächsten Ereignisses warten muss, hat nichts damit zu tun, wie lange ich darauf schon gewartet habe.

Poissonprozesse und Exponentialverteilung hängen eng zusammen. Es lässt sich nämlich zeigen, dass ein Poisson-Prozess mit Rate λ Zwischenankunftszeiten aufweist, die exponentialverteilt mit Parameter λ sind. Die mittlere Zwischenankunftszeit ergibt sich dann zu $1/\lambda$.

Zurück zum Modell

Wir nehmen also an, dass die Rechenzeit einer Transaktion pro erfolgreich erhaltener Sperre exponentialverteilt sei mit Parameter μ', also im Mittel $1/\mu'$ beträgt. Weiterhin nehmen wir an, dass die Wahrscheinlichkeit dafür, dass die Transaktion nach Bearbeitung der aktuellen Sperre terminiert, gleich Θ sei. Ist die Transaktion hingegen noch nicht beendet, so wählt sie einen bestimmten der M Bereiche zum Sperren mit Wahrscheinlichkeit

$$(1 - \Theta) \cdot \frac{1}{M}.$$

Als Konsequenz ergibt sich, dass die Zahl der Sperren geometrisch verteilt mit Parameter Θ und Mittelwert $1/\Theta$ ist. Im Mittel dauert eine erfolgreiche Transaktion demnach

$$\frac{1}{\mu'} \cdot (1 - \Theta) =: \frac{1}{\mu}.$$

Nun betrachten wir eine Transaktion T_1, die zur Zeit t aktiv ist. Das bedeutet, dass T_1 einige Bereiche gesperrt hat, aber noch nicht fertig ist. Neben T_1 seien auch die Transaktionen $T_2, T_3, \ldots, T_k$ aktiv zur Zeit t. Zu einem Konflikt kommt es, wenn T_1 nun auf einen Bereich zugreifen will, der bereits von einer der anderen aktiven Transaktionen gesperrt ist. Bezeichne $\gamma_k(t, dt)$ die Wahrscheinlichkeit dafür, dass T_1 im Intervall $[t, t + dt]$ einen Konflikt erleidet unter der Voraussetzung, dass insgesamt k Transaktionen zur Zeit t aktiv sind. Außerdem bezeichne L_i die Anzahl der Sperren von T_i zur Zeit t. Dann erhält man näherungsweise

$$\gamma_k(t,dt) = \underbrace{\mu' dt}_{\substack{\text{Rate, mit der } T_1 \text{ mit} \\ \text{der alten Sperre fertig} \\ \text{wird}}} \cdot \underbrace{(1-\Theta)}_{\substack{\text{Wahrscheinlichkeit,} \\ \text{dass eine zusätzliche} \\ \text{Sperre benötigt wird}}} \cdot \underbrace{\frac{L_2 + L_3 + \ldots L_k}{M}}_{\substack{\text{Wahrscheinlichkeit,} \\ \text{dass eine besetzte} \\ \text{Sperre benötigt wird}}}$$

Wegen

$$\frac{L_2 + L_3 + \ldots L_k}{M} \approx (k-1) \cdot \frac{1}{M} \cdot \underbrace{\frac{1}{\Theta}}_{\substack{\text{mittlere Zahl von} \\ \text{Sperren pro TA}}}$$

erhält man daraus

$$\gamma_k(t,dt) \approx \frac{\mu'(1-\Theta)}{\Theta} \frac{(k-1)}{M}\, dt.$$

Durch gewichtete Summation über alle k erhält man aus dieser bedingten Konfliktwahrscheinlichkeit die unbedingte Konfliktwahrscheinlichkeit $\gamma(t,dt)$:

$$\begin{aligned}
\gamma(t,dt) &= \sum_{k=1}^{\infty} \gamma_k(t,dt) \cdot P(k \text{ Transaktionen gleichzeitig aktiv}) \\
&= \sum_{k=1}^{\infty} \frac{\mu'(1-\Theta)}{\Theta} \frac{(k-1)}{M}\, dt \cdot P(k \text{ Transaktionen gleichzeitig aktiv}) \\
&= \frac{\mu'(1-\Theta)}{\Theta} \frac{(1)}{M}\, dt \left[\sum_{k=1}^{\infty} k \cdot P(k \text{ Transaktionen}) - \sum_{k=1}^{\infty} P(k \text{ Transaktionen}) \right] \\
&= \frac{\mu'(1-\Theta)}{\Theta} \frac{(1)}{M}\, dt \left[\bar{k} - (1 - P(k=0)) \right],
\end{aligned}$$

wobei $\bar{k}$ = die mittlere Anzahl aktiver Transaktionen beschreibt. Somit erhalten wir als stationäre Konfliktrate

$$\gamma_k(t,dt) = \frac{\mu'(1-\Theta)}{\Theta \cdot M} \cdot \left[\bar{k} - 1 + P(k=0)) \right].$$

Sei $f(T)$ die Wahrscheinlichkeit dafür, dass eine allgemeine Transaktion T Zeiteinheiten läuft, ohne von einem Konflikt betroffen zu sein. Da das Eintreten von Konflikten einen Poisson-Prozess mit Rate γ darstellt, erhält man:

$$f(T) = e^{-\gamma \cdot T}$$

und damit

$$
\begin{aligned}
p &:= P(\text{ allgemeine Transaktion ist ohne Neustart erfolgreich }) \\
&= \int_0^\infty f(h) \cdot \underbrace{P(h \leq \text{TA - Zeit} \leq h + dh)}_{\substack{= \mu \cdot e^{-\mu h}, \text{ da TA-Zeit exponentialverteilt mit Parameter } \mu}} \; dh \\
&= \mu \int_0^\infty e^{-\gamma h} \cdot e^{-\mu h} \, dh \\
&= \frac{\mu}{\mu + \gamma}.
\end{aligned}
$$

Durch Einsetzen ergibt sich schließlich

$$p = \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M} \cdot (\bar{k} - 1 + P(k = 0))}.$$

Hierbei handelt es sich um eine implizite Gleichung, da p auch noch in $\bar{k}$ steckt.

Als leicht vereinfachte untere Schranke für die Wahrscheinlichkeit, dass eine Transaktion erfolgreich ist, ohne einen Neustart durchführen zu müssen, erhalten wir also

$$p \geq \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M} \cdot \bar{k}}.$$

Verbesserung der unteren Schranke mit Little´s Result

Um bessere untere Schranken für p zu gewinnen, benötigen wir Formeln für $\bar{k}$ und $P(k = 0)$. Ein Weg hierzu führt über **Little´s Result**. Dieses besagt, dass in sehr allgemeinen Wartesystemen die mittlere Anzahl der im System befindlichen Kunden gleich dem Produkt aus Ankunftsrate und mittlerer Systemzeit (= Wartezeit + Bedienzeit) pro Kunde ist. In unserem Fall heißt das:

$$\underbrace{\bar{k}}_{\substack{\text{mittlere Transaktionszahl}}} = \underbrace{\lambda}_{\text{Ankunftsrate}} \cdot \underbrace{\overline{T}}_{\substack{\text{mittlere TA-Dauer}}}$$

Sei nun $\overline{T}_i$ die mittlere Zeit, die eine Transaktion benötigt, wenn sie genau im i-ten Versuch erfolgreich ist. Dann erhalten wir für die mittlere Transaktionsdauer

$$
\begin{aligned}
\overline{T} &= \sum_{k=1}^{\infty} \overline{T}_i \cdot P(i - 1 \text{ Neustarts}) \\
&= \sum_{k=1}^{\infty} \overline{T}_i \cdot \underbrace{(1-p)^{i-1}}_{\substack{i-1\text{-mal} \\ \text{erfolglos}}} \cdot \underbrace{p}_{\substack{i\text{-ter} \\ \text{Versuch} \\ \text{O.K.}}}
\end{aligned}
$$

Nun ist aber

$$\overline{T}_i = \frac{1}{\mu}$$

sowie $\overline{T}_i \leq i \cdot \overline{T}_1$, da erfolglose Versuche im Allgemeinen kürzer sind als erfolgreiche

$$\begin{aligned} \Rightarrow \quad \overline{T} &= \sum_{i=1}^{\infty} \overline{T}_i \cdot (1-p)^{i-1} \cdot p \leq \frac{1}{\mu} \underbrace{\sum_{i=1}^{\infty} i(1-p)^{i-1} p}_{=1/p} \\ &= \frac{1}{\mu \cdot p} \end{aligned}$$

Setzt man dies via Little´s Result in unser bisheriges Ergebnis ein, so erhält man

$$p \geq \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{p} \cdot \frac{1}{\mu}} \geq \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{\mu}. \tag{4.1}$$

Das Simple-Two-Phase-Locking

Wenn beim Simple-2-PL eine Transaktion startet, dann benötigt sie mehrere Sperren. Immer dann, wenn eine der angeforderten Sperren belegt ist, setzt die Transaktion zurück und beginnt von neuem. Dadurch stellen genaugenommen die Transaktionsankünfte keine Poisson-Prozesse mehr dar. Für unsere näherungsweisen Berechnungen soll dies aber keine Rolle spielen.

Im Allgemeinen macht eine Transaktion somit mehrere Versuche an deren Ende jeweils ein Konflikt steht, bis letztlich ein Versuch doch erfolgreich ist. In unserer bisherigen Darstellung haben wir einmal die Konfliktrate γ einer Transaktion (pro Zeiteinheit) abgeschätzt. Ferner haben wir eine untere Schranke für die Wahrscheinlichkeit p gefunden, dass eine Transaktion ungestört durchkommt.

Ist nun $M \gg 1$, dann können wir das Transaktionssystem als so genanntes $M/G/\infty$-System interpretieren. Diese Notation deutet an, dass wir ein System vor uns haben, bei dem Kunden gemäß einem Poisson-Prozess ankommen (das M steht für memoryless) und dann eine allgemeine Bedienung erfahren (G wie general), d.h. es werden keine speziellen Annahmen über die Abfertigungszeit gemacht. Außerdem sollen hierfür ∞ viele Bediener zur Verfügung stehen.

In unserem Fall liegt dann ein Ankunftsprozess der Rate λ vor, die Bedienrate beträgt $k\mu p$, ferner ist $P(K=0) = e^{-\lambda/\mu p}$.

Dann erhält man aus (4.1)

$$p \geq \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M} \left(\frac{\lambda}{\mu p} + e^{-\lambda/\mu p} - 1 \right)}.$$

Eine kurze Rechnung ergibt

$$p \geq \frac{1 - \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{\mu}}{1 + \frac{1-\Theta}{\Theta^2 M} \left(\frac{\lambda}{\mu p} + e^{-\lambda/\mu p} - 1 \right)}$$

und schließlich

$$p \geq \frac{-\frac{\lambda}{\mu}}{\log\left(1-\frac{\lambda}{\mu}\right)}, \tag{4.2}$$

falls $\frac{\lambda}{\mu} < 1$ (Stabilitätsbedingung) und $\frac{1}{\Theta} \leq \sqrt{M}$ gilt.

Ist dagegen $\frac{1}{\Theta}$ groß, z.B. $\frac{1}{\Theta} \approx M$, d.h. eine Transaktion benötigt praktisch alle Sperren, dann war die bisherige Abschätzung der Sperrenzahl

$$L_2 + L_3 + \ldots + L_k \approx \frac{k-1}{\Theta} \approx (k-1)M$$

zu pessimistisch, da es ja nicht mehr als M Sperren gibt. Schlimmstenfalls erhält man hier

$$\begin{aligned} L_2 + L_3 + \ldots + L_k &\leq M-1 \\ \Rightarrow \gamma_k(t,dt) &\leq \mu'(1-\Theta)(1-\frac{1}{M})dt \\ \Rightarrow p = \frac{\mu}{\mu+\gamma} &\geq \frac{\mu'\Theta}{\mu'\Theta + \mu'(1-\Theta)\frac{M-1}{M}} \\ &= \frac{M\Theta}{M\Theta + (1-\Theta)(M-1)} \end{aligned} \tag{4.3}$$

als untere Schranke, die nicht mehr von k abhängig ist.

Beispielergebnisse:

Abschließend wollen wir die gefundenen Abschätzungen noch graphisch veranschaulichen. Betrachten wir als erstes den Fall $1/\Theta \leq \sqrt{M}$ (kleine Transaktion) und $\rho = \lambda/\mu < 1$. Hierfür erhielten wir die Abschätzung (4.2)

$$p_L = P(\text{Transaktion kommt durch}) = \frac{-\rho}{\log(1-\rho)},$$

deren Verlauf in Abbildung 4.4 dargestellt ist.

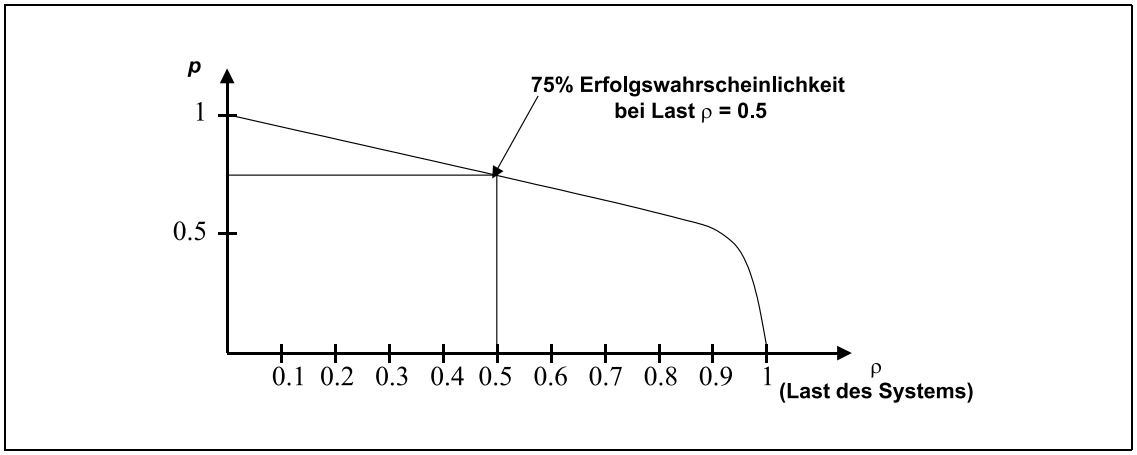


Abbildung 4.4: Leistungsverhalten von Simple-2-PL bei kleinen Transaktionen

Aus dieser Kurve lässt sich ersehen, dass im Falle kleiner Transaktionen bei nicht zu hoher Last das Simple-2-PL gar nicht mal so schlecht funktioniert, z.B. beträgt bei einer Last $\rho = 0,5$ die Chance auf ein Durchkommen ohne Störung immerhin etwa 75%, d.h. 3 von 4 Fällen sind im Mittel erfolgreich.

Bei großen Transaktionen ($1/\Theta > \sqrt{M}$) wird das Leistungsverhalten deutlich schlechter. Zur Abschätzung können wir die Ungleichungen (4.1) und (4.3) verwenden, d.h. wir haben für die Wahrscheinlichkeit p_L, dass eine Transaktion störungsfrei verläuft, die beiden unteren Schranken

$$p_{L_1} = 1 - \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{\mu}$$

und

$$p_{L_2} = \frac{M\Theta}{M\Theta + (1-\Theta)(M-1)}.$$

Nehmen wir als Beispiel ein System mit $M = 1000$ sperrbaren Bereichen und einer mittleren Zahl angeforderter Sperren pro Transaktion von $1/\Theta = 40 > \sqrt{1000}$, so erhalten wir das in Abbildung 4.5 dargestellte Ergebnis.

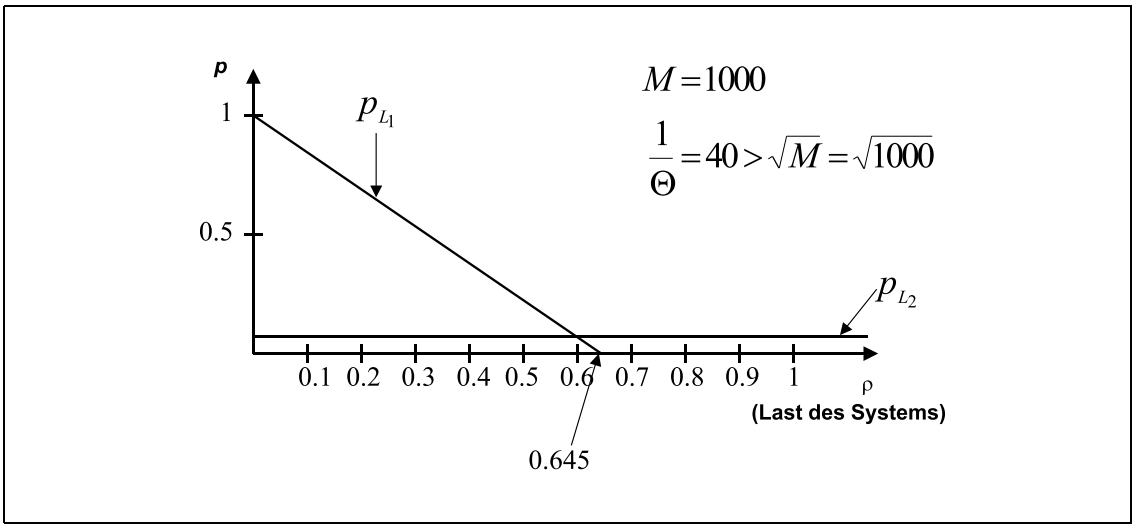


Abbildung 4.5: Leistungsverhalten von Simple-2-PL bei großen Transaktionen

Ab ca. 60% erweist sich also – bezogen auf die untere Schranke – das Simple-2-PL als so gut wie unbrauchbar.

KAPITEL 5

Deadlocks

Anfang des 20. Jahrhunderts sah sich der Gesetzgeber des US-Bundesstaates Kansas gezwungen, eine Vorfahrtsregelung an Eisenbahnkreuzungen einzuführen. Zu diesem Behufe erließ er folgende Bestimmung: „Nähern sich zwei Züge einer Kreuzung, so hat jeder von ihnen zum vollständigen Stillstand zu kommen. Ferner darf keiner von beiden weiterfahren, solange der andere noch dasteht.“[1].

Dies ist ein geradezu klassisches Beispiel für einen Systemzustand, in dem Prozesse auf Ereignisse wie z.B. die Zuteilung von Betriebsmitteln warten, die niemals eintreten werden. Einen solchen Zustand bezeichnet man als **Deadlock** oder **Verklemmung** (siehe Abbildung 5.1). Eine wichtige Aufgabe bei der Entwicklung von Betriebssystemen, die **Multiprogramming** erlauben, besteht darin, Regelungen für den Umgang mit derartigen Situationen vorzusehen.

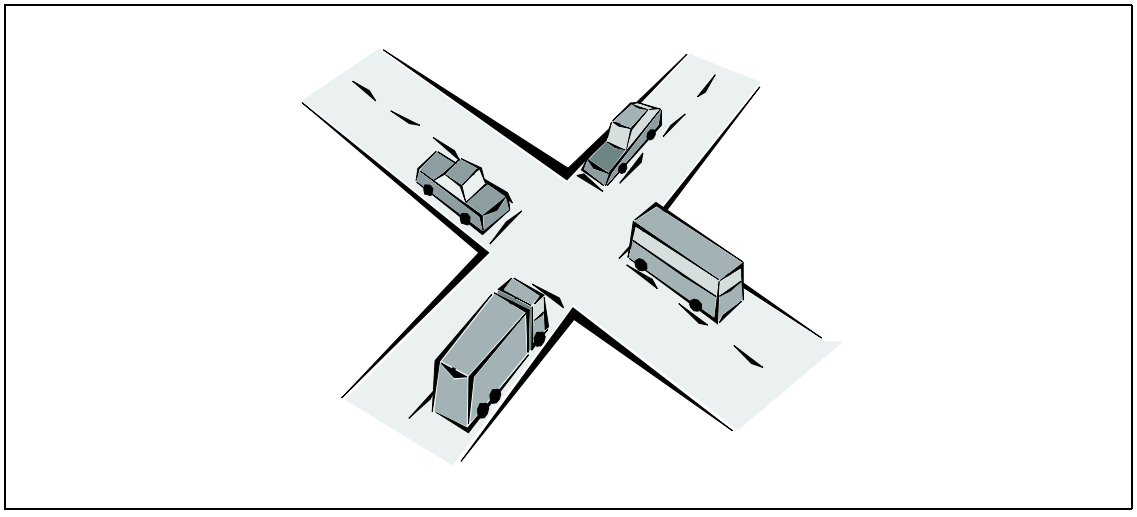


Abbildung 5.1: Das klassische Beispiel für ein Deadlock

[1]Nachzulesen in „The Columbus Chicken Statute, and More Bonehead Legislation“ von D. Hyman, S. Greene Press, Lexington, 1985

5.1 Systemmodell und notwendige Bedingungen

Betrachten wir zunächst kurz das zugrundeliegende Systemmodell. Wir nehmen an, das System bestehe aus n Prozessen $P_1, \ldots, P_n \quad (n \geq 2)$ und einer Anzahl verschiedener Betriebsmittel $B_1, \ldots, B_m$. Von jedem Betriebsmittel liegen evtl. mehrere identische Exemplare vor.

Die Prozesse fordern im Laufe ihrer Abarbeitung unterschiedliche Betriebsmittel an. Ist ein Exemplar des angeforderten Betriebsmittels gerade verfügbar, so wird es dem Prozess zugeteilt, woraufhin dieser in der Abarbeitung fortfährt und das Betriebsmittel wieder freigibt, sobald er es nicht mehr benötigt. Sind aber sämtliche Exemplare des angeforderten Betriebsmittels gerade belegt, so muss der Prozess solange warten, bis eines davon wieder frei wird.

Derartige Anforderungen und Zuteilungen von Betriebsmitteln lassen sich am besten graphisch charakterisieren. Eine Möglichkeit hierzu bietet der **Request-Allocation-Graph** (siehe Abbildung 5.2 und Abbildung 5.3).

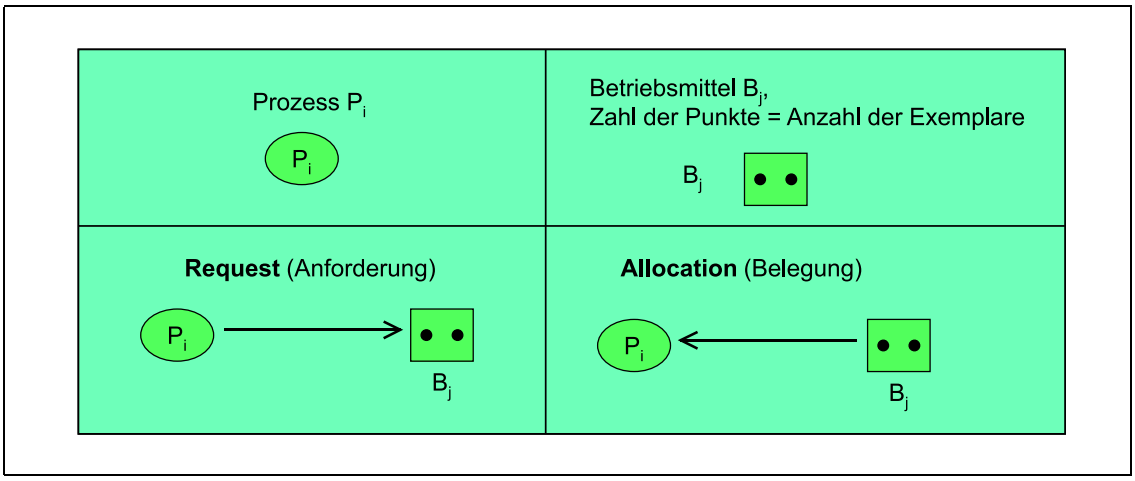


Abbildung 5.2: Erläuterung zum Request-Allocation-Graph

Ein Pfeil von einem Betriebsmittel zu einem Prozess bedeutet dabei, dass das Betriebsmittel gerade von dem Prozess belegt ist. Zeigt ein Pfeil von einem Prozess zu einem Betriebsmittel, so symbolisiert dies, dass der Prozess gerne auf das Betriebsmittel zugreifen möchte.

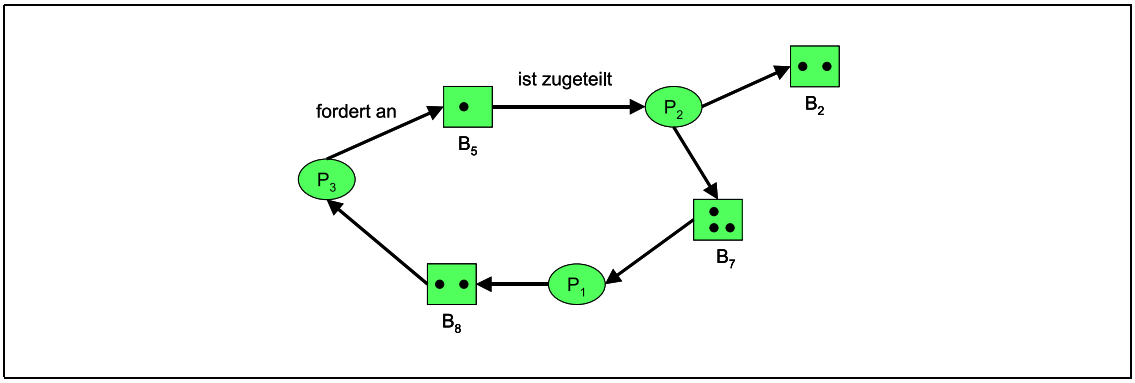


Abbildung 5.3: Beispiel eines Request-Allocation-Graphen

In unserem Beispiel in Abbildung 5.3 belegt also der Prozess P_3 gerade das Betriebsmittel

B_8, möchte aber gerne noch auf B_5 zugreifen.

Dieser Graph lässt sich dadurch vereinfachen, dass man die Betriebsmittel weglässt; man erhält so einen **Wait-for-Graphen** (siehe Abbildung 5.4).

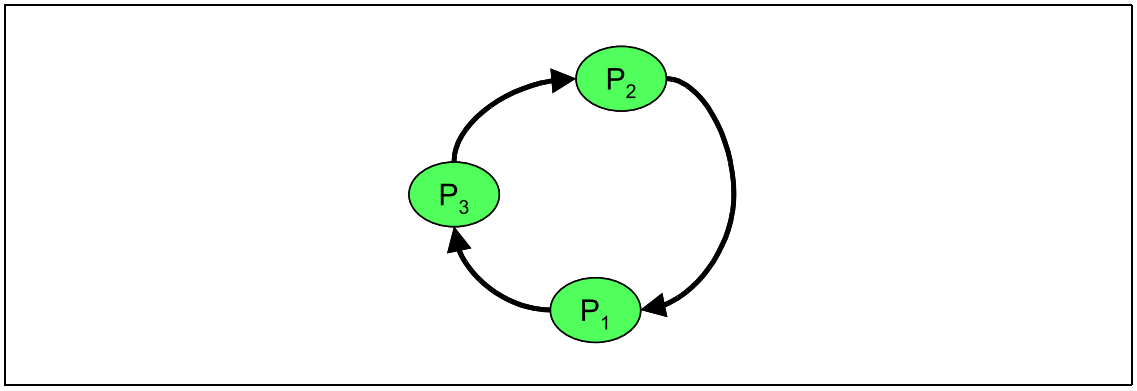


Abbildung 5.4: Beispiel eines Wait-for-Graphen

Hier bedeutet der Pfeil von P_3 nach P_2 einfach, dass P_3 ein Betriebsmittel anfordert, nämlich B_5, das von P_2 gehalten wird.

Deadlocks und Systemzustände können für zwei Betriebsmittel mit so genannten **Prozessfortschrittsdiagrammen** veranschaulicht werden. Ein Beispiel dazu ist in Abbildung 5.5 gegeben.

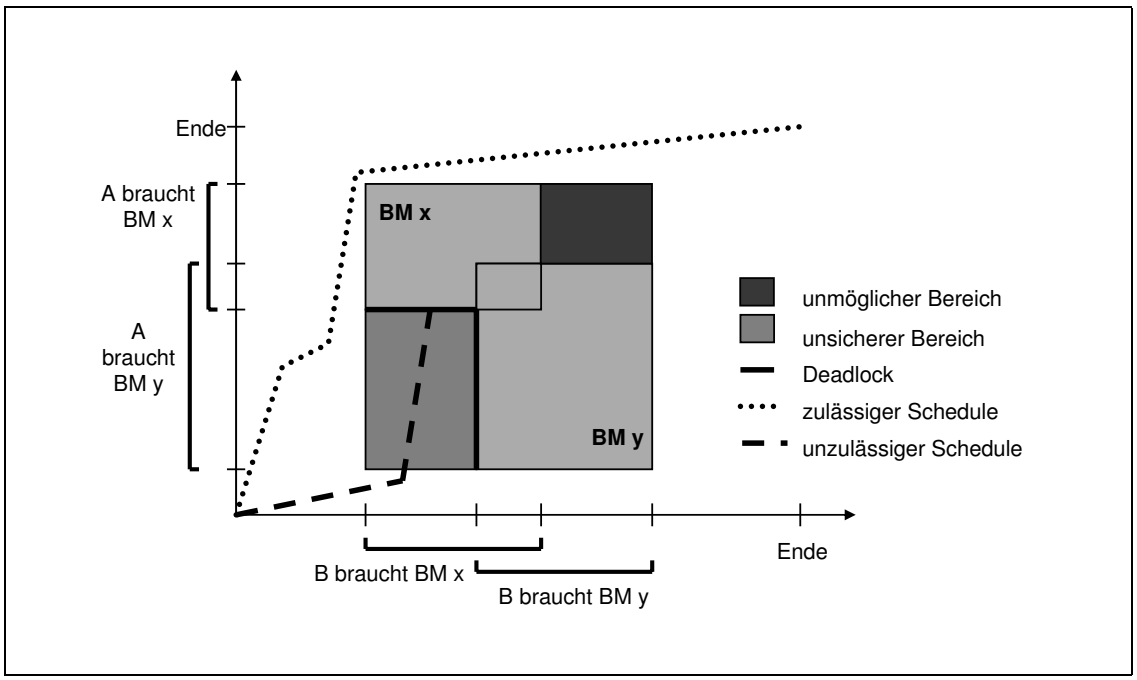


Abbildung 5.5: Prozessfortschrittsdiagramm

Im Beispiel konkurrieren zwei Prozesse P_1 und P_2 um zwei Betriebsmittel B_x und B_y. Ein **Schedule** gibt an, wie die Prozesse innerhalb ihrer Rechenzeit voranschreiten. Schraffierte Flächen geben an, in welchen Zeiträumen Betriebsmittel von beiden Prozessen beansprucht werden. Diese Bereiche können durch einen Schedule nicht erreicht werden (**unmöglicher Bereich**). Gelangt ein Schedule in einen **unsicheren Bereich**, so erreicht er bei weiterem Fortschritt zwangsläufig einen **Deadlock-Zustand**.

Das Vorliegen eines Deadlocks lässt sich durch die folgenden vier Bedingungen charakterisieren, die allesamt gleichzeitig erfüllt sein müssen:

1. **Circular-Wait:** Es muss im Request-Allocation-Graphen (bzw. im Wait-for-Graphen) eine geschlossene Kette vorliegen. Genauer gesagt muss eine Menge $P_1, \ldots, P_k$ wartender Prozesse existieren derart, dass P_1 auf ein Betriebsmittel wartet, das gerade von P_2 belegt wird. P_2 seinerseits auf eines wartet, das von P_3 belegt wird, bis schließlich P_k auf ein Betriebsmittel wartet, das von P_1 belegt wird.
2. **Exclusive-Use:** Keines der fraglichen Betriebsmittel kann von mehreren Prozessen gleichzeitig benutzt werden (kein **Betriebsmittel-Sharing**).
3. **Hold-and-Wait:** Prozesse können ein neues Betriebsmittel anfordern, ohne dabei die bisher von ihnen belegten zurückzugeben.
4. **No-Preemption:** Betriebsmittel können von den Prozessen nur freiwillig zurückgegeben werden, ein Entzug von außen (durch höhere Gewalt) ist nicht möglich.

Natürlich muss man im Ernstfall, wenn also einmal ein Deadlock-Zustand erreicht ist, bei der letzten Bedingung ansetzen, d.h. von außen entweder einen Prozess „killen“ oder ihm einige Betriebsmittel wegnehmen.

5.2 Gegenmaßnahmen

Prinzipiell gibt es drei verschiedene Möglichkeiten, mit der Deadlock-Problematik umzugehen. Man könnte erstens sicherstellen, dass das System niemals in einen Deadlock-Zustand geraten kann. Dies erreicht man entweder durch **Deadlock-Prevention**, d.h. Ausschluss einer der vier notwendigen Bedingungen oder durch **Deadlock-Avoidance**, hierunter versteht man Verfahren, die Deadlocks durch Zusatzinformationen vermeiden; beispielsweise kann man vor jeder „riskanten“ Operation erst einmal einen Test machen und die Operation nur dann zulassen, wenn sie „ungefährlich“ ist. Eine zweite Möglichkeit besteht darin, Deadlocks zuzulassen. In diesem Fall sind Mechanismen nötig, die in der Lage sind, das Vorliegen eines Deadlock-Zustands zu erkennen und ihn daraufhin zu beseitigen. Drittens schließlich gibt es die Möglichkeit, die Problematik zu ignorieren. Hier nimmt man also an, dass Deadlocks so gut wie nie auftreten. Interessanterweise verfahren die meisten Betriebssysteme (einschließlich UNIX) nach dieser Methode.

5.2.1 Deadlock-Prevention

Wie bereits erwähnt, lassen sich Deadlocks einfach dadurch ausschließen, dass man eine der Bedingungen 1 - 4 unmöglich macht. Entsprechend gibt es folgende vier Ansätze:

- **Betriebsmittel-Sharing:** um Bedingung 2 zu vereiteln. Leider geht dies häufig aus praktischen Gründen nicht, z.B. den Drucker zu teilen.
- **Preemption:** d.h. Eingriffe von außen sind möglich. Dies hat leicht den Ruch einer Gewaltmaßnahme.
- **Alles oder nichts:** Ein Prozess geht nicht nach dem Prinzip „hold and wait“ vor, sondern fordert alle Betriebsmittel gleichzeitig an. Dieses Verfahren besticht offensichtlich durch seine Ineffizienz. Man beginnt den Bau eines Hauses ja in der Regel auch nicht erst dann, wenn man endlich die Tapeten eingekauft hat.

- **Unmöglichkeit eines Zyklus:** Dies wird algorithmisch häufig durch Einführung einer Betriebsmittel-Hierarchie folgendermaßen verwirklicht:

  Teile Betriebsmittel in Klassen $K_1, \ldots, K_h$ ein (mit $h \geq 1$) (siehe Abbildung 5.6).

  - Wenn ein Prozess P_i aktuell Betriebsmittel der Klasse K_r (und niedrigerer Klassen) belegt, dann darf er „nur nach oben“ anfordern, d.h. er darf nur auf Betriebsmittel der Klassen $K_{r+1}, \ldots, K_h$ warten.
  - Sollte er dagegen weitere Betriebsmittel einer niedrigeren Klasse K_i (mit $i \leq r$) benötigen, dann muss er zunächst alle derzeit belegten Betriebsmittel der Klassen $K_i, K_{i+1}, \ldots, K_r$ freigeben.

  Dass dieses Schema einen Zyklus vermeidet, lässt sich leicht nachweisen (siehe Abbildung 5.6).

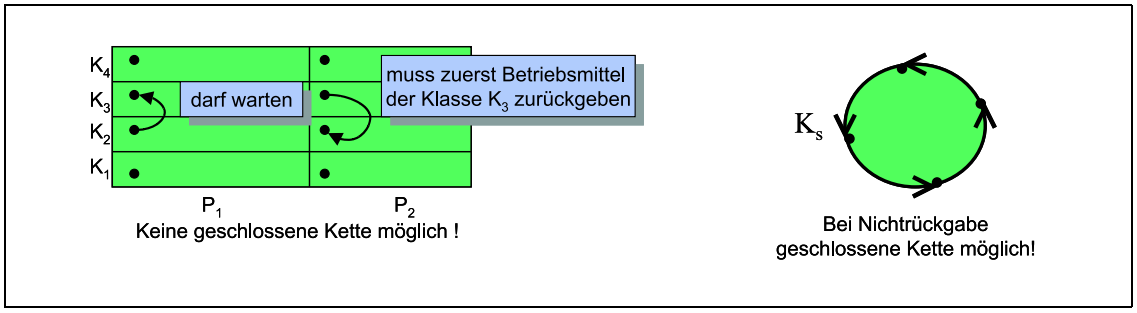


Abbildung 5.6: Betriebsmittelhierarchie

  Nehmen wir hierzu an, es existiere ein Zyklus $(P_0, P_1, \ldots, P_n)$ aufeinander wartender Prozesse. Bezeichne zu jedem dieser Prozesse $F(P_i)$ die höchste Klasse der von ihm angeforderten Betriebsmittel. Dann gilt: $F(P_0) < F(P_1) < F(P_2) < \ldots < F(P_n) < F(P_0)$, da jeder Prozess ja nur Betriebsmittel einer höheren als den bisher belegten Klassen anfordern darf. $F(P_0) < F(P_0)$ ist aber ein Widerspruch.

Wichtig ist, dass auch bei Anforderung eines zusätzlichen Betriebsmittels einer bereits gehaltenen Klasse die „alten“ Betriebsmittel dieser Klasse erst zurückgegeben werden müssen.

Lässt man in dem angegebenen Schema nur eine einzige Klasse zu ($h = 1$), so ist dies äquivalent zur sequenziellen Nutzung der Betriebsmittel, da in diesem Fall vor einer Neuanforderung erst alle bisher belegten Betriebsmittel zurückzugeben sind.

5.2.2 Deadlock-Avoidance

Die bislang vorgestellten Mechanismen verhindern das Eintreten eines Deadlocks, indem sie sicherstellen, dass nicht alle vier notwendigen Bedingungen gleichzeitig erfüllt sein können. Sie erreichen dies durch Einschränkungen bei der Anforderung von Betriebsmitteln. Leider wird dabei die Deadlockfreiheit durch Ineffizienz bei der Betriebsmittelnutzung erkauft, was letztlich auf Kosten des Systemdurchsatzes geht.

Eine alternative Vorgehensweise zur Vermeidung von Deadlocks nutzt zusätzliche Informationen, um zu prüfen, ob Zustandsübergänge aufgrund von Betriebsmittelzuteilungen sicher sind. Es soll also ausgeschlossen werden, dass ein System, das sich nachgewiesenermaßen in einem gefahrlosen Zustand befindet, durch die Zuteilung eines Betriebsmittels in einen Zustand gerät, der evtl. in einen Deadlock münden könnte.

Was ein **sicherer Zustand** ist, darüber gehen die Meinungen teilweise auseinander. Wir halten uns an folgende Festlegung:

DEFINITION

Sicherer Zustand

Ein System, bestehend aus den Prozessen $P_1, \ldots, P_n$, befindet sich genau dann in einem sicheren Zustand, wenn eine Reihenfolge von Prozessausführungen (P_{i1}, $P_{i2}, \ldots, P_{in}$) existiert, sodass für alle j der Prozess P_{ij} weitergeführt werden kann, wenn P_{ij} seine maximalen Anforderungen stellt und wenn alle vorherigen Prozesse $P_{i1}, \ldots P_{ij-1}$ ihre Betriebsmittel zurückgegeben haben.

Existiert keine solche Folge von Prozessen (siehe Abbildung 5.7), so ist das System in einem **unsicheren** Zustand. Der Startzustand ist natürlich automatisch **sicher**.

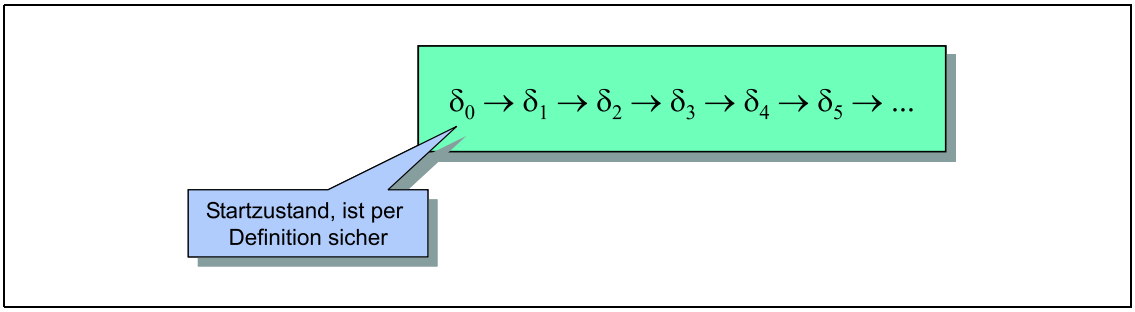


Abbildung 5.7: Folge von sicheren Zuständen

Ein sicherer Zustand lässt sich also nach dieser Definition dadurch charakterisieren, dass es eine Möglichkeit gibt, alle Anforderungen auch im ungünstigsten Fall zu befriedigen. Betrachten wir zur Verdeutlichung folgendes Beispiel:

BEISPIEL

Beispiel einer möglichen Betriebsmittelanforderungsreihenfolge für 3 Prozesse

Es gebe ein Betriebsmittel, davon aber gleich 12 Exemplare. Die drei Prozesse P_1, P_2 und P_3 haben folgende maximale Anforderungen: $Max(P_1) = 10$, $Max(P_2) = 4$ und $Max(P_3) = 9$ Exemplare des Betriebsmittels. Die aktuelle Zuteilung betrage $Alloc(P_1) = 5$, $Alloc(P_2) = 2$ und $Alloc(P_3) = 2$. Es gilt nun, eine Reihenfolge für die drei Prozesse zu finden, die die oben genannte Bedingung erfüllt.

Durch einfaches Ausprobieren stellt man fest, dass die Reihenfolge P_2, P_1, P_3 folgende Zustandsübergänge impliziert:

Schritt	Prozess P_1	Prozess P_2	Prozess P_3	freie Betriebsmittel	ausgeführte Aktion
1	5	2	2	3	2 BM an P_2
2	5	4	2	1	4 BM von P_2
3	5	0	2	5	5 BM an P_1
4	10	0	2	0	10 BM von P_1
5	0	0	2	10	7 BM an P_3
6	0	0	9	3	9 BM von P_3
7	0	0	0	12	

Folglich befindet sich das System in einem sicheren Zustand, kann also nicht in einen Deadlock geraten. Dieses Beispiel wird in Abbildung 5.8 nochmal graphisch dargestellt.

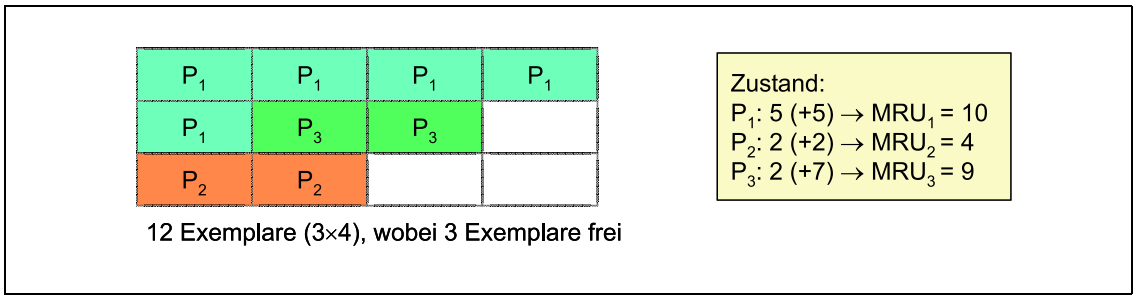


Abbildung 5.8: Visualisierung des obigen Beispiels

Es muss aber nicht immer einen Deadlock-freien Schedule existieren. Wir betrachten das obige Beispiel erneut mit einer anderen Anfangssituation.

BEISPIEL

Betriebsmittelanforderungsreihenfolge, die zu einem Deadlock führt

Nehmen wir jetzt an, die aktuelle Zuteilung von P_3 betrage 3 statt 2, $Alloc(P_3) = 3$. Dann erhalten wir:

Schritt	Prozess P_1	Prozess P_2	Prozess P_3	freie Betriebsmittel	ausgeführte Aktion
1	5	2	3	2	2 BM an P_2
2	5	4	3	0	4 BM von P_2
3	5	0	3	4	

Da auch keine andere Reihenfolge der Prozessabarbeitung möglich ist, stellt also der Zustand $Alloc(P_1) = 5$, $Alloc(P_2) = 2$ und $Alloc(P_3) = 3$ einen unsicheren Zustand dar. Er kann zu einem Deadlock führen, nämlich dann, wenn P_1 und P_3 ihre Maximalanforderungen nutzen.

Deadlock-Avoidance-Algorithmen machen sich dieses Konzept nun dadurch zunutze, dass sie vor einer geplanten Betriebsmittelzuteilung prüfen, ob sich das System dadurch evtl. in einen unsicheren Zustand manövriert. Wird durch diesen Test festgestellt, dass der Folgezustand unsicher wäre, dann wird die geplante Zuteilung verworfen, andernfalls wird sie ausgeführt.

5.2.3 Der Banker´s Algorithmus

Ein bekannter Algorithmus zur Deadlock-Avoidance ist der **Banker´s Algorithmus**. Er verfährt analog zu einem vorsichtigen Bankier, der sichergehen will, dass er die monetären Forderungen seiner Kunden mittels der verliehenen (und zu gegebener Zeit zurückkommenden) und der im Tresor verbliebenen Geldmenge befriedigen kann.

Man kann den Banker´s Algorithmus für zwei unterschiedliche Aufgaben heranziehen: Einmal dient er zur Deadlock-Avoidance, nämlich dann, wenn er auf maximalen Anforderungen beruht oder zur Erkennung von Deadlocks; dann basiert er auf dem aktuellen Zustand.

Für die Implementierung des Algorithmus gehen wir von n Prozessen $P_1, P_2, \ldots, P_n$ und m Betriebsmitteltypen $B_1, B_2, \ldots, B_m$ aus. Seien ferner:

$Q_{j,s}^{max}(k)$:= zur Zeit k von P_j maximal zusätzlich anforderbare Betriebsmittel vom Typ B_s
$Q_{j,s}(k)$:= zur Zeit k von P_j aktuell angeforderte Betriebsmittel vom Typ B_s
$H_{j,s}(k)$:= zur Zeit k von P_j gehaltene Betriebsmittel vom Typ B_s
$V_s(k)$:= zur Zeit k verfügbare Betriebsmittel vom Typ B_s

Der Einfachheit halber betrachten wir ab sofort Q_j^{max}, Q_j, H_j und V als m-dimensionale Vektoren, entsprechend gilt dann das =-Zeichen Komponentenweise.

Wann gilt nun ein Zustand als unsicher? Im Fall der Deadlock-Avoidance muss man einen Zustand dann als unsicher bezeichnen, wenn es eine Teilmenge D von Prozessen gibt, sodass für keinen Prozess aus D seine maximale Anforderung erfüllbar ist, selbst dann nicht, wenn alle nicht in D enthaltenen Prozesse ihre gehaltenen Betriebsmittel zurückgeben.

Formal ausgedrückt ist ein Zustand zum Zeitpunkt k genau dann unsicher, wenn gilt:

$$\exists D \subseteq \{P_1, P_2, \ldots, P_n\} \text{ mit } Q_j^{max}(k) > V(k) + \sum_{r \notin D} H_r(k) \quad \forall j \in D$$

Um den Banker´s Algorithmus zur Erkennung von Deadlocks einsetzen zu können, genügt es, Q_j^{max} durch Q_j zu ersetzen.

Doch zurück zur Deadlock-Avoidance. Der Banker´s Algorithmus dafür besteht aus zwei Teilen: der **Sicherheitsprüfung** und der **Betriebsmittelzuteilung** an einen geeigneten Prozess. Er lautet folgendermaßen:

Teil 1: Sicherheitsprüfung

1. Durchsuche $P_1, \ldots, P_n$ nach dem ersten unmarkierten Prozess P_i mit $Q_j^{max}(k) \leq V(k)$
   Existiert kein solches i, dann gehe nach 3.

2. Setze $V(k) := V(k) + H_i(k)$, d.h. P_i arbeitet mit den Betriebsmitteln und gibt die gehaltenen danach zurück;
   Markiere P_i;
   Zurück nach 1.

3. Wurden alle P_i markiert, dann ist das System bezüglich des Betriebsmittelvektors B sicher, andernfalls ist es unsicher.

Teil 2: Betriebsmittel-Zuteilung an einen Prozess P_j

1. Ist $Q_j(k) \leq V(k)$, dann gehe nach 2.;
   ansonsten muss P_j warten.

2. Setze (probehalber)
   $V(k) := V(k) - Q_j(k)$
   $H_j(k) := H_j(k) + Q_j(k)$
   $Q_j^{max}(k) := Q_j^{max}(k) - Q_j(k)$

3. Prüfe (mit Teil 1), ob der neue Zustand sicher ist.
   wenn ja: O.K.
   wenn nein: Zuteilung verwerfen (restore)

Exemplarisch wollen wir diesen Algorithmus auf unser obiges Beispiel anwenden:

BEISPIEL

Banker´s Algorithmus für die korrekte BM-Anforderung

	P_1	P_2	P_3	ausgeführte Aktion
$Q_j(k)$				
				1.) Prüfe $P_1 : Q_1^{max}(k) = 5 > 3 = V(k) \Rightarrow$ geht nicht
$H_j(k)$	5	2	2	2.) Prüfe $P_2 : Q_2^{max}(k) = 2 \leq 3 = V(k) \Rightarrow$
				$V(k) = V(k) + H_2(k) = 3 + 2 = 5$
$Q_j^{max}(k)$	5	2	7	P_2 markieren
$V(k)$		3		
				1.) Prüfe $P_1 : Q_1^{max}(k) = 5 \geq 5 = V(k)$
$V(k)$		5		$V(k) = V(k) + H_1(k) = 5 + 5 = 10$
				P_1 markieren
$V(k)$		10		1.) Prüfe $P_1 : Q_3^{max}(k) = 7 < 10 = V(k)$
				$V(k) = V(k) + H_3(k) = 10 + 2 = 12$
				P_3 markieren
$V(k)$		12		

BEISPIEL

Banker´s Algorithmus für die unkorrekte BM-Anforderung

	P_1	P_2	P_3	ausgeführte Aktion
$Q_j(k)$				
				1.) Prüfe $P_1 : Q_1^{max}(k) = 5 > 2 = V(k) \Rightarrow$ geht nicht
$H_j(k)$	5	2	3	2.) Prüfe $P_2 : Q_2^{max}(k) = 2 \leq 2 = V(k) \Rightarrow$
				$V(k) = V(k) + H_2(k) = 2 + 2 = 4$
$Q_j^{max}(k)$	5	2	6	P_2 markieren
$V(k)$		2		
				1.) Prüfe $P_1 : Q_1^{max}(k) = 5 \geq 4 = V(k) \Rightarrow$ geht nicht
$V(k)$		4		2.) Prüfe $P_3 : Q_3^{max}(k) = 6 \geq 4 = V(k) \Rightarrow$ geht nicht
				$\Rightarrow D = \{P_1, P_3\}$ nicht bearbeitbar; Zustand unsicher

Laufzeitverhalten des Banker's Algorithmus

Abschließend noch eine Bemerkung zur Laufzeit des Algorithmus (siehe auch Abbildung 5.9). In Prozesstests lässt sich ermitteln, dass diese im besten Fall linear in der Anzahl der Prozesse n wächst, während sie im „Worst-Case“ $O(n^2)$ beträgt. Letzteres lässt sich leicht plausibel machen, wenn man davon ausgeht, dass im ersten Schritt alle n unmarkierten Prozesse zu testen sind, da erst der n-te passt. Im nächsten Schritt kann es dann im ungünstigsten Fall zu $n-1$ Tests kommen, danach zu $n-2$ usw. In der Summe ergibt dies:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \text{ Tests.}$$

Eine Verbesserung (aber nur für den Fall, dass $m = 1$ ist, also lediglich ein Betriebsmitteltyp vorliegt) liefert das **Verfahren von Holt**. Es beruht darauf, dass zunächst

Banker's Algorithmus	Banker's Algorithmus in Kombination mit dem Verfahren von Holt
Laufzeit im „worst case“: $\approx \frac{n^2}{2} + \frac{n}{2}$	Laufzeit im „worst case“: a) Sortierung z.B mit Heap-Sort $= O(n \cdot \log n)$ b) lineare Testzeit $= O(n)$
$\Rightarrow O(n^2)$	$\Rightarrow O(n \cdot \log n)$
Beispiel: $n = 1000$	
$\Rightarrow \approx \frac{1000^2}{1000} + \frac{1000}{2}$	$\Rightarrow \approx 1000 \log 1000$
$\approx \frac{10^6}{2}$	≈ 10.000 Operationen
$= 500.000$ Operationen	

Abbildung 5.9: Laufzeit des Banker´s Algorithmus für $m = 1$ mit und ohne dem Verfahren von Holt

die Betriebsmittelanforderungen der Größe nach vorsortiert werden. Daraufhin werden die Prozesse der Reihe nach von der kleinsten bis zur größten Anforderung abgetestet. In diesem Fall wächst die Laufzeit für den Banker´s Algorithmus nur noch linear in n. Berücksichtigt man noch, dass ein schnelles Verfahren für die Vorsortierung (wie z.B. Heapsort) $O(n \cdot \log(n))$ aufweist, so ergibt das Verfahren von Holt insgesamt eine Laufzeitabhängigkeit der Größenordnung $O(n \cdot \log(n))$, was für große n eine enorme Verbesserung darstellen kann.

Natürlich verbraucht der Banker´s Algorithmus Rechenzeit. Daher ist die Frage, wie oft man ihn sinnvollerweise aufrufen sollte, durchaus ernst zu nehmen. Es gibt darauf nur einige heuristische Antworten:

- Häufige Aufrufe sind zu befürworten, wenn zu befürchten ist, dass Deadlocks häufig auftreten. Der Grund liegt darin, dass nach dem Eintreten eines Deadlocks, der oft eigentlich nur einige wenige Prozesse betrifft, ein Risiko dafür besteht, dass sich andere Prozesse ebenfalls verklemmen (und es zu mehreren Deadlock-Ketten kommt).
- Warten, bis die Systemleistung offensichtlich stark heruntergeht, und erst danach zu prüfen. Dieses Vorgehen beruht auf der Annahme, dass Deadlocks nicht eintreten, solange die Systemleistung (wie etwa der Durchsatz von Jobs pro Zeiteinheit) genügend hoch ist.
- Prüfungen in periodischen Abständen durchführen. Prüft man hierbei zu selten, so besteht allerdings die Gefahr, dass das System u.U. lange stillsteht, prüft man zu häufig, so wird evtl. unnötig Rechenzeit vergeudet.
- Schließlich besteht eine letzte Alternative darin, die Gefahr eines Deadlocks einfach

zu ignorieren und erst zu reagieren (dann freilich mit Brachialgewalt), wenn wirklich gar nichts mehr geht. Diese Strategie ist zwar schlecht, aber einfach zu realisieren.

Es stellt sich noch die Frage, was zu tun ist, wenn wirklich einmal ein Deadlock eingetreten ist. Grundsätzlich sind hier zwei Vorgehensweisen möglich. Zum einen kann man einfach hergehen und sämtliche verklemmte Prozesse killen und neustarten. Eine etwas differenziertere Methode besteht darin, der Reihe nach einzelne der betroffenen Prozesse zu killen und darauf zu hoffen, dass sich von irgendeinem Punkt an der Deadlock löst, da eine der vier notwendigen Bedingungen durchbrochen wurde.

KAPITEL 6

CPU-Scheduling

In einem **Einprozessorsystem**, das **Multiprogramming** erlaubt, läuft in der Regel einer der Prozesse auf der CPU und befindet sich folglich im Zustand *running*, während verschiedene andere Prozesse im *ready*-Status darauf warten, bis der genannte Prozess die CPU freigibt. Ist dies der Fall, so stellt sich natürlich die Frage, welcher der lauffähigen Prozesse vom Betriebssystem in den running-Zustand versetzt werden soll, also als nächster drankommt.

Eine Antwort darauf kann von verschiedenen Kriterien abhängig gemacht werden. Einmal kann man versuchen, möglichst *fair* vorzugehen, also z.B. keinen Prozess zu lange warten zu lassen oder keinem Prozess zu viel CPU-Zeit zuzuteilen. Man kann den Prozessen verschieden hohe Wichtigkeit zuordnen, was auf eine **Prioritätenregelung** hinausläuft. Oder man kann danach trachten, das Leistungsverhalten der CPU in den Mittelpunkt zu stellen, um beispielsweise einen möglichst hohen Durchsatz, hohe CPU-Auslastung, eine minimale mittlere Wartezeit (= Zeit, bis ein Prozess drankommt) oder eine minimale mittlere Antwortzeit (= Wartezeit + Bedienzeit) zu gewährleisten. Im Allgemeinen wird man sich nicht auf ein einziges Kriterium versteifen, sondern verschiedene Aspekte in einer Mischung betrachten. Aufgrund dieser Mischung muss man sich dann für eine **Scheduling-Strategie** entscheiden.

6.1 Die Schedulingstrategien FIFO und LIFO

Am einfachsten ist die Strategie **First-In-First-Out** (FIFO) bzw. **First-Come-First-Served** (FCFS). Nach diesem Schema erhält der Prozess, der als erster CPU-Zeit angefordert hat als nächster Zugriff auf die CPU. Daher lässt sich die Prozessreihenfolge ganz einfach mittels einer FIFO-Warteschlange verwalten. Gravierende Nachteile dieser Strategie liegen in der Tatsache begründet, dass **Langzeitjobs** in einem gewissen Sinne bevorzugt werden, d.h. die Werte für mittlere Wartezeiten, Antwortzeiten oder etwa die Anzahl wartender Kunden fallen bei FIFO oft schlechter aus als bei anderen Strategien.

Machen wir uns das an einem Beispiel deutlich, indem wir drei Prozesse mit Rechenzeit $P_1 = 24$, $P_2 = 3$ und $P_3 = 3$ betrachten. Nehmen wir an, dass P_2 unmittelbar nach Beendigung von P_1 beginnt (analog P_3), d.h. wir vernachlässigen evtl. auftretende Umschaltzeiten. Das Schedule für dieses Beispiel ist in Abbildung 6.1 dargestellt. Diese graphische Darstellung wird **Gantt-Chart** genannt.

Die Berechnung der mittleren Wartezeit $\overline{t}$ ist nicht schwer und ergibt:

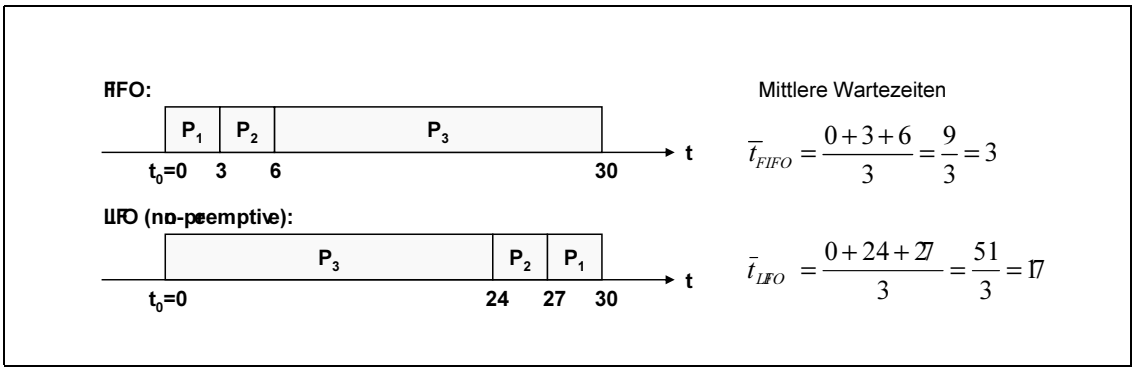


Abbildung 6.1: Gantt-Chart für FIFO und LIFO

$$\bar{t}_{FIFO} = \frac{0 + 24 + 27}{3} = \frac{51}{3} = 17$$

Vergleichen wir dieses Ergebnis mit einer anderen, ähnlich einfachen Strategie, nämlich **Last-In-First-Out** (LIFO). Es ist unschwer zu erraten, dass die Vorgehensweise hierbei darin besteht, denjenigen Prozess als ersten auf die CPU zugreifen zu lassen, der als letzter seinen Anspruch geltend gemacht hat. Der Gantt-Chart zu LIFO ist auch in Abbildung 6.1 dargestellt.

Kurze Rechnung ergibt für die mittlere Wartezeit:

$$\bar{t}_{LIFO} = \frac{0 + 3 + 6}{3} = 3$$

Also eine deutlich kürzere mittlere Wartezeit für unser Beispiel. Im Allgemeinen aber und das heisst auf lange Sicht und für beliebige Schedules sind die mittleren Wartezeiten von FIFO und LIFO gleich. Man kann sogar weitergehen und feststellen, dass im Schnitt die mittleren Wartezeiten aller beliebigen Strategien für beliebige Schedules gleich sind, solange wir Strategien betrachten, die ihre Auswahl nicht aufgrund der Jobdauern treffen und bei denen die Abarbeitung **work conserving** ist, d.h. das Umschalten zwischen Prozessen ist vernachlässigbar klein.

Ein Kriterium, für das sich auch bei langfristiger allgemeiner Betrachtung ein Unterschied zwischen FIFO und LIFO feststellen lässt, ist die Varianz (oder Streuung) der Wartezeiten.

Hierzu eine kurze Bemerkung: In der Wahrscheinlichkeitsrechnung betrachtet man oft die Ergebnisse zufälliger Messungen, die in der Regel um einen Mittelwert schwanken. Um diese Schwankung zu charakterisieren, berechnet man die durchschnittliche quadratische Abweichung der Zufallsereignisse vom Mittelwert und nennt dies die Varianz V der Zufallsgröße. Nimmt also eine Messgröße bei verschiedenen Messungen nur Werte an, die nahe an ihrem Mittelwert liegen, so ist die Varianz klein, während eine große Varianz vorliegt, wenn die einzelnen Messungen ziemlich unterschiedliche Ergebnisse erbringen. Eine genauere Einführung findet sich in jedem Buch über Wahrscheinlichkeitsrechnung.

Beispielsweise ist im obigen Beispiel:

$$V_{FIFO} = \frac{1}{3}\left((0-17)^2 + (24-17)^2 + (27-17)^2\right) = 146$$

und

$$V_{LIFO} = \frac{1}{3}\left((0-3)^2 + (3-3)^2 + (6-3)^2\right) = 6$$

Allgemein kann man nun sagen, dass die Varianz der Wartezeit bei LIFO echt kleiner ist als die Varianz der Wartezeit bei FIFO. Der Unterschied verstärkt sich dabei noch mit wachsender CPU-Auslastung.

6.2 Die Schedulingstrategien SJF und SRPT

Ein allgemeines Problem bei FIFO bzw. LIFO besteht in der Benachteiligung, die kurzlaufende Jobs durch Langläufer erfahren. Im täglichen Leben trifft man auf dieses Problem beispielsweise auch an Supermarktkassen. Die pragmatische Lösung in diesem Fall bestand in der Einführung von Schnellkassen, die nur Kunden abfertigen, die maximal $10 \pm \varepsilon$ Artikel kaufen möchten.

Eine Verallgemeinerung dieser Idee führt zur Strategie **Shortest-Job-First** (SJF) auch **Shortest-Processing-Time-First** (SPT) genannt. Dabei muss man voraussetzen, dass die Dauer, die ein Job für seine Bearbeitung benötigt, im Voraus bekannt ist. Unter dieser Voraussetzung bedient man dann einfach zu jedem Zeitpunkt den aktuell vorliegenden kürzesten Job.

Man kann zeigen, dass die mittlere Wartezeit bei SJF kleiner oder gleich der mittleren Wartezeit jeder anderen nichtunterbrechenden Strategie ist. Eine **nichtunterbrechende** (**non-preemptive**) Strategie lässt sich hierbei dadurch charakterisieren, dass ein Prozess, dessen Bearbeitung einmal begonnen wurde, dann auch an einem Stück bis zu seiner Terminierung fortgeführt wird. Strategien, die die Unterbrechung eines einmal begonnenen Jobs erlauben, heissen dementsprechend **unterbrechend** bzw. **preemptiv**. Offensichtlich ist die Menge aller non-preemptiven Strategien eine Untermenge aller preemptiven Strategien. FIFO, LIFO und SJF sind allesamt non-preemptive Strategien.

BEISPIEL

FIFO, LIFO und SPT im Vergleich

Es lässt sich leicht an einem Beispiel demonstrieren, dass bezüglich der mittleren Wartezeit SPT in dieser Klasse optimal ist. Hierzu seien vier Prozesse P_1 bis P_4 mit den Dauern $P_1 = 6$, $P_2 = 8$, $P_3 = 7$ und $P_4 = 3$ gegeben. Die Ergebnisse sind in Abbildung 6.2 dargestellt.

Optimal in der Klasse aller preemptiven Strategien ist eine Variante von SJF, nämlich die Strategie **Shortest-Remaining-Processing-Time-First** (SRPT) (siehe Abbildung 6.3). Sie unterscheidet sich von SJF insofern, als ein Prozess, der sich gerade in Bearbeitung befindet, (evtl. sogar mehrmals) unterbrochen werden kann, sobald ein neuer Prozess ankommt, dessen Bearbeitungszeit noch kürzer ist als die Restzeit, die der gerade rechnende Job noch benötigt.

Es lässt sich leicht zeigen, dass SRPT optimal bezüglich der mittleren Wartezeit unter allen Strategien ist, die **work conserving** sind also vernachlässigbare Umschaltzeiten aufweisen. Ein Beispiel zu SRPT ist in Abbildung 6.4 dargestellt.

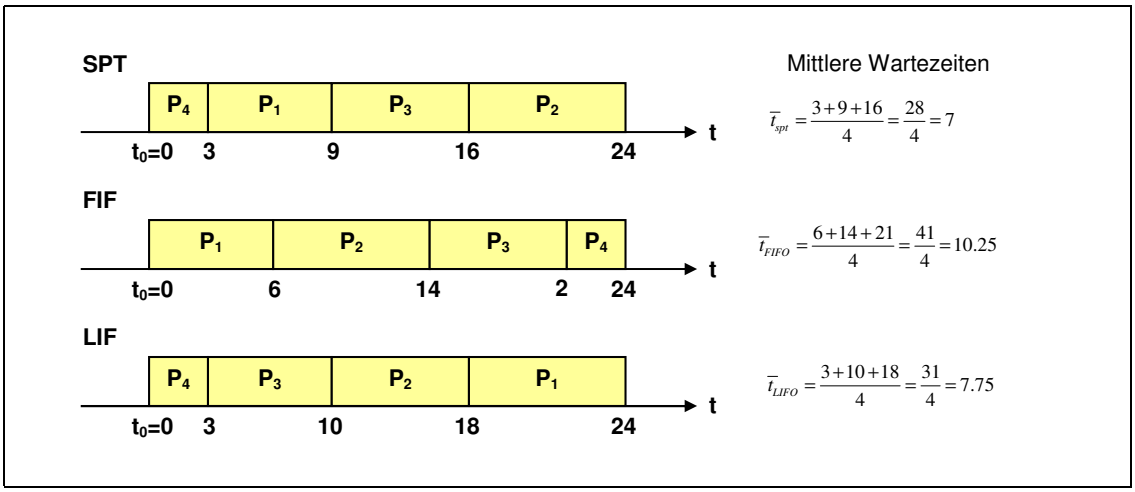


Abbildung 6.2: Die Strategien SJF, FIFO und LIFO im Vergleich

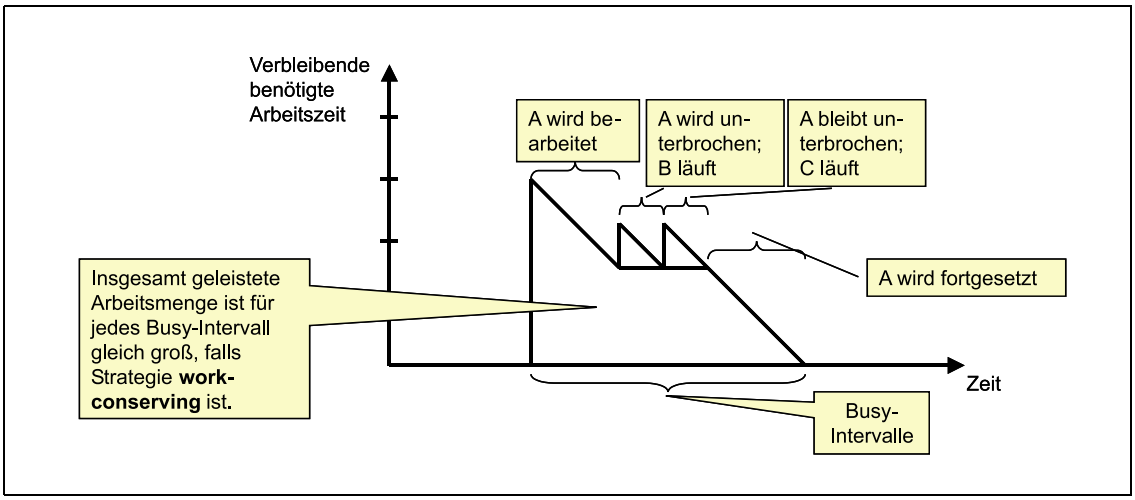


Abbildung 6.3: Die SRPT-Strategie

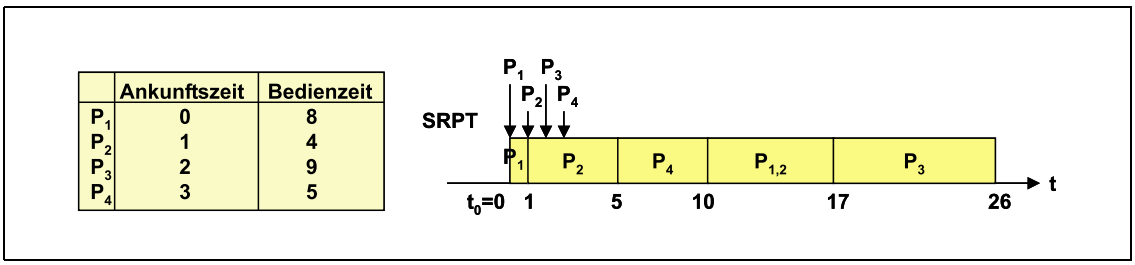


Abbildung 6.4: Beispiel zu SRPT

6.3 Die Schedulingstrategien SEPT und SERPT

Bei der Realisierung von SJF bzw. SRPT trifft man auf ein grundlegendes Problem: Woher weiß man a priori, wie lange ein Job dauern wird? Dieser Frage kann man auf zwei Arten begegnen.

Erster Ansatz: Man erwartet eine Angabe der Prozessdauer durch den Nutzer, z.B. in der Art „Mein Job dauert 2 Minuten“. Der Nutzer wird sich in der Regel bemühen,

eine möglichst zutreffende Angabe zu machen, denn nennt er eine zu kurze Zeit, so kann es ihm passieren, dass sein Job vor Terminierung abgewürgt wird. Ist die Zeit, die er nennt, zu lang, so beginnt die Abarbeitung seines Jobs u.U. unnötig spät, da es kürzere gibt, deren Bearbeitung vorgezogen wird.

Zweiter Ansaz: Man versucht, die zu erwartende Prozessdauer aus Erfahrungen der Vergangenheit möglichst zutreffend zu schätzen. Hierbei nimmt man an, dass die einzelnen Jobs von verschiedenen Kundenquellen erzeugt werden. Weiß man nun, wie lange in der Vergangenheit Jobs der verschiedenen Quellen gedauert haben, so kann man daraus mehr oder weniger gut auf das Verhalten in der Zukunft schließen (siehe Abbildung 6.5).

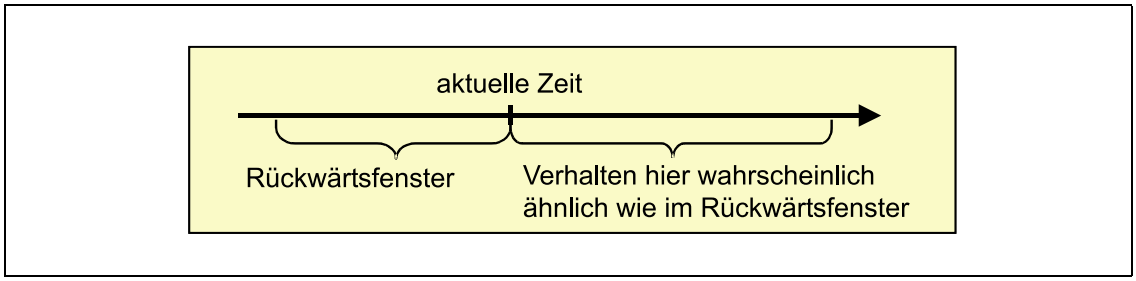


Abbildung 6.5: Lernen aus der Vergangenheit

Diese Überlegungen führen zu Strategien wie **Shortest-Expected-Processing-Time-First** (SEPT) bzw. **Shortest-Expected-Remaining-Processing-Time-First** (SERPT).

Einfache Schätzverfahren verwenden hierbei etwa den Mittelwert aus zurückliegenden Fenstern also z.B. den Durchschnitt der letzten n Jobs der betreffenden Quelle oder beispielsweise 80% des Maximalwerts der letzten n Jobs o.ä.

Eine andere Idee, um adaptiv aus der Vergangenheit zu lernen, verwendet das **Exponential-Averaging**. Sei hierzu τ_n die Schätzung für den n-ten Job einer bestimmten Kundenquelle, sowie t_n die tatsächliche Dauer dieses Jobs. Dann erhält man eine Schätzung für den $(n+1)$-ten Job nach der Formel:

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha)\tau_n$$

Der Parameter α liegt dabei zwischen 0 und 1 und beeinflusst die Art des Lernens aus der Vergangenheit. Dies sieht man am besten, wenn man die beiden Extremfälle betrachtet:

- $\alpha = 0$: $\tau_{n+1} = \tau_n = \ldots = \tau_0$: Die Schätzung bleibt immer dieselbe; Lernen findet nicht statt.

- $\alpha = 1$: $\tau_{n+1} = t_n$: hier wird jeweils nur der allerletzte Zustand für die Schätzung herangezogen, die Vergangenheit davor interessiert nicht $\Rightarrow$ völlig hektisches Verhalten.

Bessere Kompromisse verwenden ein α echt zwischen 0 und 1. Dies bewirkt letztlich ein exponenzielles Abklingen der Vergangenheit. Führt man die Rekursion nämlich explizit

aus, so erhält man schließlich:

$$\begin{aligned}
\tau_{n+1} &= \alpha \cdot t_n + (1-\alpha) \cdot \tau_n \\
&= \alpha \cdot t_n + (1-\alpha) \cdot (\alpha \cdot t_{n-1} + (1-\alpha) \cdot \tau_{n-1}) \\
&= \alpha \cdot t_n + (1-\alpha) \cdot \alpha \cdot t_{n-1} + (1-\alpha)^2 \cdot \tau_{n-1} \\
&= \ldots \\
&= \alpha \cdot t_n + (1-\alpha) \cdot \alpha \cdot t_{n-1} + \cdots + (1-\alpha)^n \cdot t_0 + (1-\alpha)^{n+1} \cdot \tau_0
\end{aligned}$$

Je größer dabei α ist, desto schneller vergisst man die Vergangenheit. Wählt man also α zu groß, so wirken sich kurzfristige Schwankungen sehr stark aus, und es findet kaum eine Glättung statt. Ist α andererseits zu klein, so erfolgt die Korrektur bei wirklichen Trendveränderungen möglicherweise viel zu langsam.

Exkurs: Digitale Sprachübertragung

Exponential-Averaging bei Scheduling-Strategien versucht, der Kurve der tatsächlich benötigten CPU-Zeit von Prozessen eines Kunden zu folgen. Ganz ähnlich muss beim Digitalisieren von Sprachsignalen der Frequenzkurve mit Abtastwerten gefolgt werden.

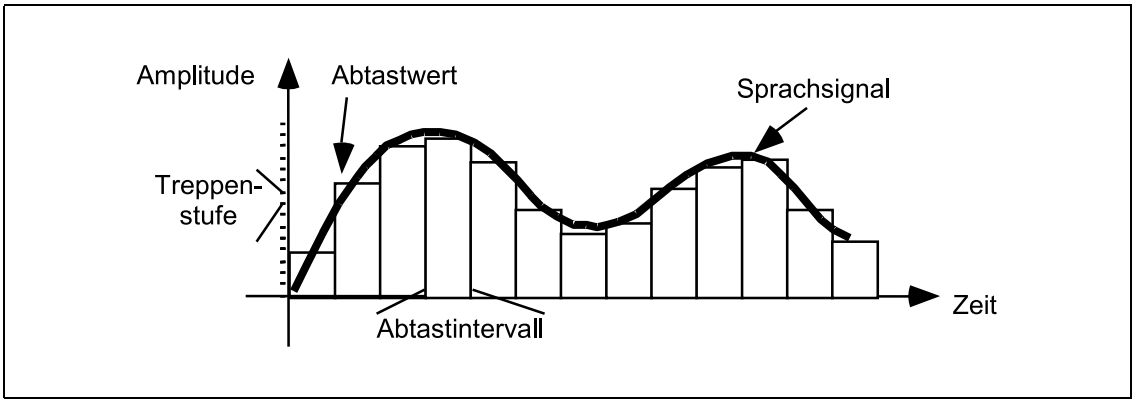


Abbildung 6.6: Digitalisieren von Sprachsignalen

Wie oft ein Signal abgetastet werden muss, geht aus dem Nyquist-Theorem hervor:

Nyquist-Theorem: Vorausgesetzt, dass ein Signalwert exakt digitalisiert werden kann, lässt sich die Originalkurve aus den digitalen Signalen genau dann wiederherstellen, wenn die Abtastrate (Abtastfrequenz) mindestens doppelt so hoch ist wie die maximale im Signal vorkommende Frequenz.

Bei Telefonen liegt die (Audio-)Signalfrequenz üblicherweise zwischen 300 und 3.400 Hertz. Eine Abtastfrequenz von ungefähr 6,8 kHz wäre demnach bei exakter Abtastung ausreichend.

Pulse Code Modulation (PCM) tastet mit einer Frequenz von 8 kHz ab und kodiert jeden Abtastwert mit 8 Bit (nicht exakt, jedoch für das menschliche Ohr ausreichend). Es müssen also alle 125 μs 8 Bit übertragen werden, was einer Übertragungsrate von 64 kBit/s entspricht[1].

[1] Man beachte, dass z.B. ein ISDN-B-Kanal genau 64 kBit/s Kapazität aufweist.

Um diese Datenrate zu senken, können z.B. die Abtastintervalle vergrößert werden (siehe Abbildung 6.6). Eine andere Möglichkeit besteht in der so genannten **Delta-Modulation**. Dabei wird nicht jeder Abtastwert, sondern nur der Startwert und danach jeweils die Änderung (Delta) zum vorherigen Wert übertragen[2]. Der Signalverlauf wird also beim Sender und Empfänger nachgeführt.

Um dieses Verfahren anwenden zu können, müssen zwei Dinge festgelegt werden:

- Die Größe der Treppenstufe, d.h. die Änderung, die ± Delta verursacht, und
- die Anzahl von Bits, mit denen Delta kodiert wird, d.h. wie viele Treppenstufen auf einmal genommen werden können.

Beispielsweise funktioniert dies bei Kodierung mit einem Bit wie folgt (siehe Abbildung 6.7). Übertrage entweder 0 (alter Wert + Delta) oder 1 (alter Wert - Delta), je nachdem welcher dieser Werte näher am aktuellen Abtastwert liegt. Die Einstellung der Treppenstufe ist jedoch schwierig. Ein kleines Delta erkennt zwar kleine Amplitudenunterschiede, kann jedoch nicht schnell auf große Schwankungen reagieren (entspricht analog einem kleinen α bei Exponential Averaging). Bei einem großen Delta-Wert (großes α bei Exponential Averaging) ist dies genau umgekehrt.

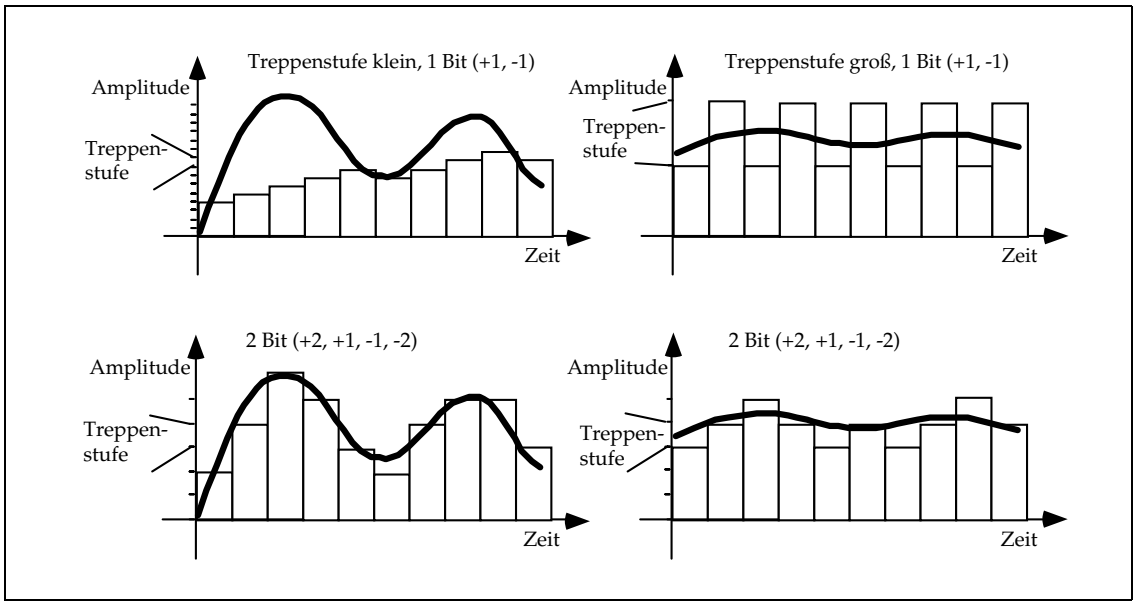


Abbildung 6.7: Delta-Modulation mit jeweils 1-Bit-Delta und 2-Bit-Delta

Offensichtlich ist ein 1-Bit-Delta sehr häufig nicht in der Lage, dem Kurvenverlauf zu folgen. Alternativ können 2-, 3- oder 4-Bit-Deltas verwendet werden (siehe Abbildung 6.7). Solche Kodierungen erhöhen andererseits auch wieder die Datenrate.

6.4 Das Priority-Scheduling

Beim **Priority-Scheduling** wird jedem Job eine Priorität zugeordnet, die Abarbeitung erfolgt dann nach dem Schema **Highest-Priority-First** (HPF) (siehe Abbildung 6.8). Eine verfeinerte Variante ordnet jeden Job in eine von mehreren Prioritätsklassen ein.

[2]Solche Strategien finden übrigens auch bei Videokompressionsverfahren wie MPEG Anwendung.

Vorrang hat dann jeweils die höchste nichtleere Prioritätsklasse, wobei innerhalb der Klasse dann z.B. nach FIFO verfahren wird. Dieses Schema kann sowohl nicht-preemptiv als auch preemptiv implementiert werden. In letzterem Fall wird ein Job mit niedriger Priorität unterbrochen, sobald ein Job höherer Priorität ankommt.

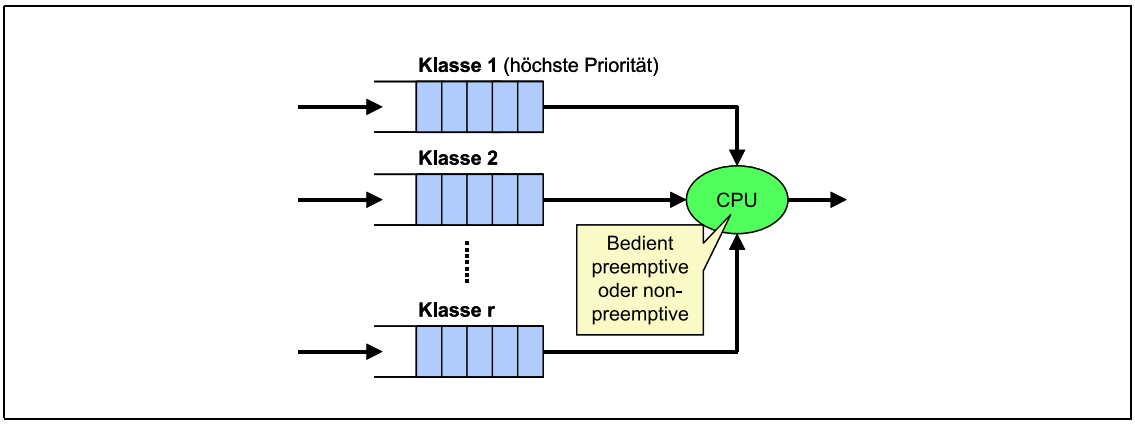


Abbildung 6.8: Priority Scheduling

Dieses Vorgehen ist natürlich insofern problematisch, als die höheren Klassen ohne weiteres die niedrigeren monopolisieren können, wobei dann im Extremfall ein Job, der sich in der niedrigsten Klasse befindet, ewig bis zu seiner Abarbeitung warten muss.

6.5 Die Schedulingstrategien Round-Robin und Processor-Sharing

Eine weitere Scheduling-Variante ist das **Round-Robin**-Verfahren (RR). Hierbei wird in gewisser Weise ein Ausgleich zwischen Langläufern und Kurzläufern unter den Jobs versucht. Die Grundidee beim Round-Robin versucht, die Antwortzeit eines Jobs näherungsweise proportional zu seiner Dauer zu gestalten.

Hierzu vergibt der Scheduler der Reihe nach an jeden Prozess ein Quantum Q – auch Zeitscheibe genannt (siehe Abbildung 6.9). Wird ein Prozess innerhalb von Q fertig, dann ist alles wunderbar. Schafft er dies nicht, so unterbricht er seine Arbeit und reiht sich ganz normal an die letzte Stelle einer FIFO-Schlange ein, und der nächste Prozess erhält nun seine Zeitscheibe Q.

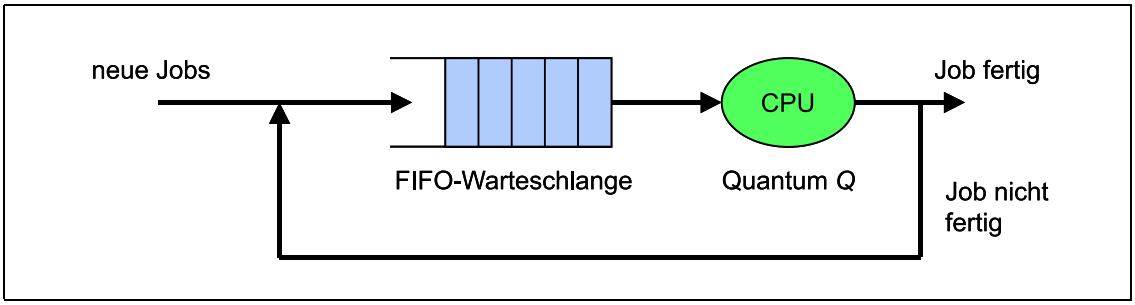


Abbildung 6.9: Round-Robin

Für den Extremfall $Q \to 0$ also für sehr kleine Quanten und wenn Jobunterbrechungen nichts kosten, erhält man daraus näherungsweise die Strategie **Processor-Sharing** (PS). Hierbei erhält dann jeder der n aktuell im System befindlichen Prozessen genau $1/n$ der CPU-Leistung. Natürlich ist zu bedenken, dass in der Praxis bei zu kleinen Quanten der

Verwaltungsoverhead schnell zu groß wird. Betrachtet man auf der anderen Seite den Fall $Q \to \infty$ also sehr große Quanten, so geht das Schema in das ganz gewöhnliche FIFO-Scheduling über.

6.6 Das Multilevel-Feedback-Queueing

Ein letztes Scheduling-Verfahren, das hier vorgestellt werden soll, ist das **Multilevel-Feedback-Queueing**. Diese Strategie arbeitet sowohl mit Prioritätsklassen (Multilevel) als auch mit Zeitscheiben. Die Idee ist dabei folgende: Ein neu ankommender Job wird in die höchste Prioritätsklasse eingereiht. Die Abarbeitung erfolgt dann mit einer **Round-Robin**-Strategie, wobei die einzelnen Prioritätsklassen unterschiedlich große Quanten erhalten. Klassen hoher Priorität haben kleine Quanten, Klassen niedrigerer Priorität entsprechend größere. War ein Job in seiner Prioritätsklasse dran, wurde allerdings nicht innerhalb der entsprechenden Zeitscheibe fertig, so wandert er in die nächstniedrigere Klasse (siehe Abbildung 6.10).

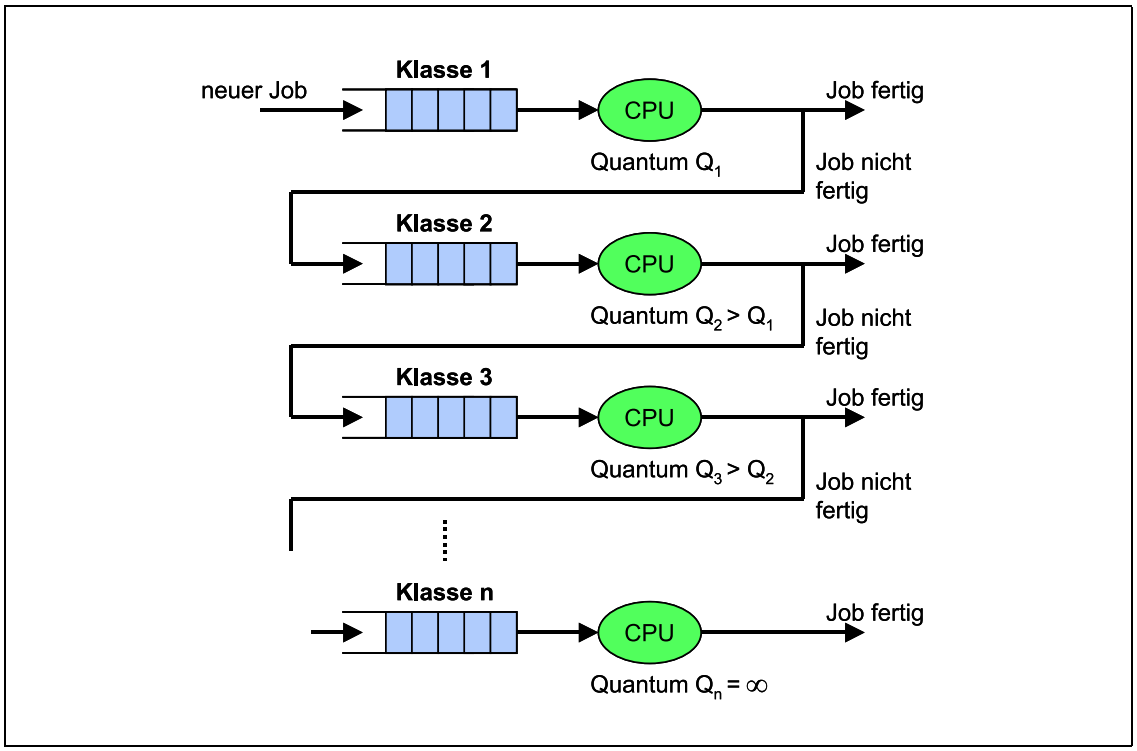


Abbildung 6.10: Multilevel-Feedback-Queueing

Bedient wird bei freier CPU stets der älteste Job der höchsten Klasse. Dies hat zur Folge, dass extrem kurze Jobs immer sofort erledigt werden können. Längere Jobs wandern in die nächste Klasse und können dadurch von Kurzläufern überholt werden. Die Strategie nimmt natürlich in Kauf, dass sehr lange Jobs u.U. auch sehr, sehr lange im System bleiben, bis sie endlich abgearbeitet werden können. Im Hochlastfall kann es wiederum zu einer Monopolisierung des Rechensystems durch Kurzläufer kommen.

TEIL III

Speicherverwaltung

KAPITEL 7

Hauptspeicherverwaltung

In den vorangegangenen Abschnitten wurde davon ausgegangen, dass die CPU durch verschiedene Prozesse gleichzeitig genutzt werden kann. Durch diese Eigenschaft wird zum einen die Leistung des Rechners bzw. Systems erhöht, zum anderen die Antwort- bzw. Bearbeitungszeit von Prozessen gesenkt. Dieser Vorteil hat jedoch auch seinen Preis: es müssen verschiedene Prozesse im Speicher verwaltet werden, d.h. der Speicher muss gemeinsam genutzt werden.

7.1 Speicherorganisation

Ein Prozesssystem enthält viele gleichzeitig aktive Prozesse, die alle unterschiedliche Anforderungen, insbesondere unterschiedlichen Speicherbedarf, haben. Jeder Prozess hat den Wunsch, schnell bedient zu werden, was unter anderem einen schnellen Speicherzugriff erfordert.

Die Probleme bei der Realisierung eines schnellen Speicherzugriffs für viele Prozesse liegen darin, dass

- schnelle Speicher sehr teuer sind (viel teurer als langsame Hintergrundspeicher wie z.B. Festplatten),
- die Speicherausnutzung schlecht ist, wenn die Programme vollständig in schnellen kleinen Speichern gehalten werden, und
- die Speicherstruktur für den Programmierer transparent sein soll.

Lösen lassen sich diese Probleme durch eine Speicherhierarchie in Kombination mit dem Konzept des virtuellen Speichers.

7.1.1 Speicherhierarchie

Im einfachsten Fall sind zwei Hierarchieebenen vorhanden, nämlich der Hauptspeicher und der Hintergrundspeicher. Der **Hauptspeicher** ist von der Kapazität her begrenzt, wohingegen der **Hintergrundspeicher** beliebig groß sein kann. Schaltet man zwischen Hauptspeicher und CPU noch einen schnellen Pufferspeicher (**Cache**), so ergibt sich folgendes Bild:

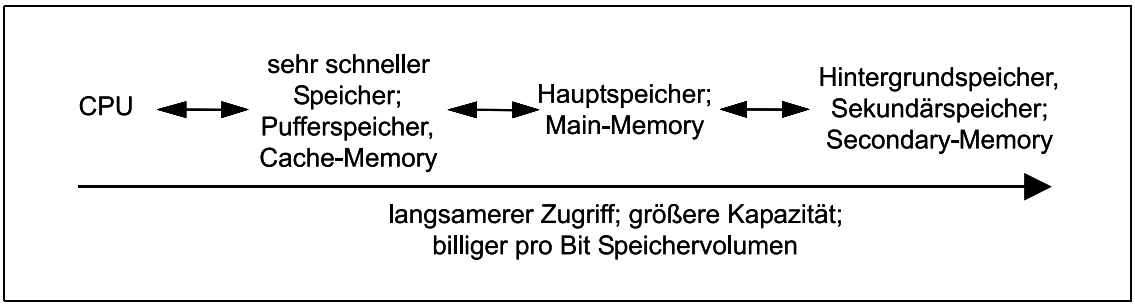


Abbildung 7.1: Hierarchie-Level eines Speichers

Bei einer solchen hierarchischen Organisation werden aktuell benötigte Daten direkt verfügbar gehalten, die anderen Daten werden in den Hintergrund verdrängt und nur bei Bedarf in den Hauptspeicher geholt.

7.1.2 Virtueller Speicher

Das Prinzip des **virtuellen Speichers** besteht darin, nur Teile des Programms im Hauptspeicher zu halten und den (beliebig großen) Rest im Hintergrundspeicher zu verwalten. Es ist also *virtuell* mehr Hauptspeicher vorhanden als *reell*. Man benötigt für solch einen virtuellen Speicher drei Bestandteile:

1. den physikalischen Adressraum eines Programms,
2. den logischen Adressraum eines Programms und
3. einen Tauschmechanismus zwischen 1. und 2.

In älteren Computersystemen organisierte der Programmierer die Zuteilung (Laden, Verdrängen) von Speicherblöcken selbst. Heute ist dies eine Aufgabe, die vom Betriebssystem automatisch übernommen wird.

Probleme der Speicherzuteilung treten an unterschiedlichsten Stellen auf. Zunächst ist es wichtig zu wissen, welche Speichereinheiten zu ersetzen sind. Dies können feste Einheiten wie zum Beispiel ganze Seiten sein, variable Größen wie Segmente und auch Mischformen von beiden. Hinsichtlich der Speicherkapazität, die im Hauptspeicher pro Programm reserviert werden soll, kann sowohl eine feste als auch eine variable Zuteilung vorgenommen werden. Weitere Probleme bestehen darin, wann Seiten oder Segmente nachgeladen werden sollen, wohin das Nachladen erfolgt und auch, was dafür ersetzt werden soll.

Die Beschäftigung mit diesen Fragen führt uns zu den Konzepten von **Segmentierung** und **Paging** (d.h. Seitenersetzung). Diese beiden Möglichkeiten können auch kombiniert werden, was in der Praxis jedoch keine große Bedeutung besitzt. Eine weitere Möglichkeit der Speicherverwaltung sind **Buddy-Systeme**, die im Weiteren Sinne einen Spezialfall der Segmentierung darstellen.

7.2 Segmentierung

Bei Fragen der Speicherverwaltung ist immer im Auge zu behalten, dass die logische Nutzersicht auf den Speicher stets eine andere ist, als die momentane Struktur des physikalischen Speichers. Aus diesem Grund sollte die oft komplexe Struktur des physikalischen

Speichers vor dem Benutzer verborgen werden können (Transparenz). Das Betriebssystem überträgt dann die Nutzersicht auf die physikalische Speicherbelegung, d.h. es muss eine Abbildung zwischen logischem und physikalischem Speicher erfolgen. Segmentierung ist ein Speicherverwaltungsschema, das versucht, die Nutzersicht möglichst direkt auf den Speicher zu übertragen.

7.2.1 Das Prinzip der Segmentierung

Der logische Adressraum ist in diesem Zusammenhang eine Sammlung von Segmenten, wobei wir unter einem **Segment** eine als logische Einheit betrachtete Informationsmenge verstehen, die aus einem oder mehreren Unterprogrammen oder aus Daten zu einem Programm bestehen kann.

Jedes Segment besitzt einen Namen und eine Länge. Adressen bestehen aus dem **Segmentnamen** und dem so genannten **Offset** innerhalb des Segments. Der Offset ist dabei der Abstand zwischen dem Segmentanfang und einer beliebigen Stelle im Segment. Der Einfachheit halber können Segmente auch durchnummeriert werden. Dieses Verfahren macht die Benennung durch Namen überflüssig. In einem solchen Fall besteht eine logische Adresse aus dem geordneten Paar (Segmentnummer, Offset).

Für jeden Prozess lädt das Betriebssystem die Segmente, die für das Fortschreiten benötigt werden, in den Hauptspeicher. Falls nicht alle Segmente der einzelnen Prozesse gleichzeitig im Hauptspeicher gehalten werden können, wird ggf. ein Nachladen bzw. Verdrängen von Segmenten erforderlich. Als Konsequenz erhält man im Hauptspeicher Belegtbereiche (mit den Segmenten) und Lücken. Benachbarte Lücken werden zu einer großen Lücke vereinigt. Das Betriebssystem muss eine Liste der Segmente und Lücken mit ihren jeweiligen Größen verwalten.

7.2.2 Segmentierungsstrategien

Wo werden nun neue Segmente im Speicher abgelegt? Zuerst wird eine Lücke gesucht, die mindestens so groß wie das Segment ist. Dort wird das Segment dann linksbündig platziert. Bezüglich der Lückenauswahl gibt es unterschiedliche Strategien:

First-Fit (FF): Platziere das Segment in die erste passende Lücke.

Best-Fit (BF): Platziere das Segment in die kleinste passende Lücke.

Worst-Fit (WF): Platziere das Segment in die größte passende Lücke.

Rotating-First-Fit (RFF): Wie First-Fit, jedoch wird von der Position der vorherigen Platzierung (hier: Ende der benutzten Lücke) ausgehend die nächste passende Lücke gesucht.

Der Nachteil von First-Fit ist der hohe Verschnitt durch die Bildung vieler kleiner Lücken am Anfang des Speichers. Bei Best-Fit wirkt sich nachteilig aus, dass in manchen Fällen der verbleibende Rest einer Platzierung extrem klein und später praktisch nicht mehr verwendbar ist. Durch Simulationen konnte gezeigt werden, dass Rotating-First-Fit in der Regel am besten abschneidet. Unter bestimmten Voraussetzungen gilt im statistischen Mittel (unter Definition von $>$ als „besser als“) die folgende Relation:

$$\text{RFF} > \text{FF} > \text{BF} > \text{WF}$$

Man kann jedoch für alle Verfahren Szenarien erstellen, in denen jede der Strategien im Einzelfall optimal oder besonders schlecht ist. Dies soll an den folgenden Beispielen – ohne Berücksichtigung der Strategie Worst-Fit – erläutert werden.

Beispiel zur Unbrauchbarkeit von Best-Fit

Als Ausgangssituation stehen zwei Lücken der Größen 900 und 500 zur Verfügung. Es sollen Segmente der Größen 400, 450 und 500 platziert werden. Insgesamt ist also genügend Speicherplatz vorhanden, um die Anforderungen zu erfüllen. Jedoch ermöglichen nur die Verfahren First-Fit bzw. Rotating-First-Fit (von vorne begonnen) eine Platzierung der Anforderungen.

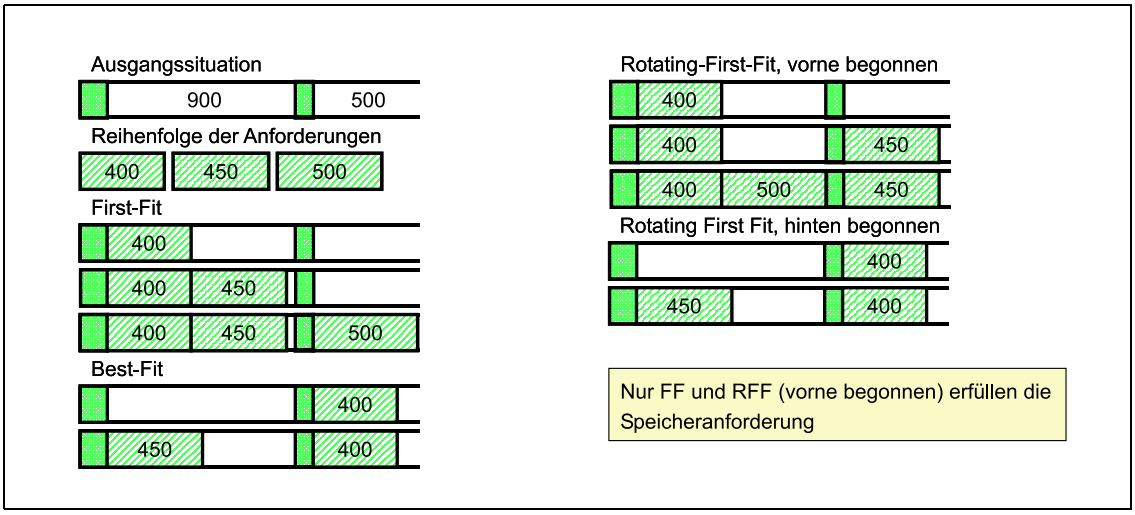


Abbildung 7.2: Unbrauchbarkeit von Best-Fit

Abbildung 7.2 stellt diesen Sachverhalt noch einmal grafisch dar. Während First-Fit und Rotating-First-Fit (von vorne begonnen) die Möglichkeit bieten, alle drei Segmente zu speichern, ist dies bei Best-Fit und Rotating-First-Fit (von der letzten Lücke aus begonnen) nicht der Fall.

Beispiel zur Unbrauchbarkeit von First-Fit und Rotating-First-Fit

In diesem Fall stehen in Abbildung 7.3 als Ausgangssituation Speicherbereiche der Größe 700, 500 und 800 zur Verfügung. Es treten Anforderungen der Größen 400, 600 und 800 auf. Jedoch kann lediglich das Best-Fit-Verfahren die Anforderungen platzieren. In diesem Fall erweist sich also Best-Fit als die bessere Alternative.

First-Fit und auch Rotating-First-Fit führen beide zu einem ungenügenden Ergebnis, da bei ausreichendem Speicherplatz jeweils Teile von Segmenten verschenkt werden.

Im Zusammenhang mit der Segmentierung ergibt sich die Frage, wie weit man einen segmentierten Speicher füllen darf. Ein zu niedriger Füllgrad bewirkt zwar ein einfaches Platzieren, jedoch ist der Speicher schlecht ausgelastet. Hingegen gewährleistet ein hoher Füllgrad eine gute Speicherauslastung, nur ist aufgrund der vielen kleineren Lücken (externe Fragmentierung) und dem damit verbundenen Suchaufwand die Platzierung von neuen Segmenten sehr aufwendig.

Eine Alternative besteht darin, nachträglich Daten im Speicher umzuordnen (**Garbage-**

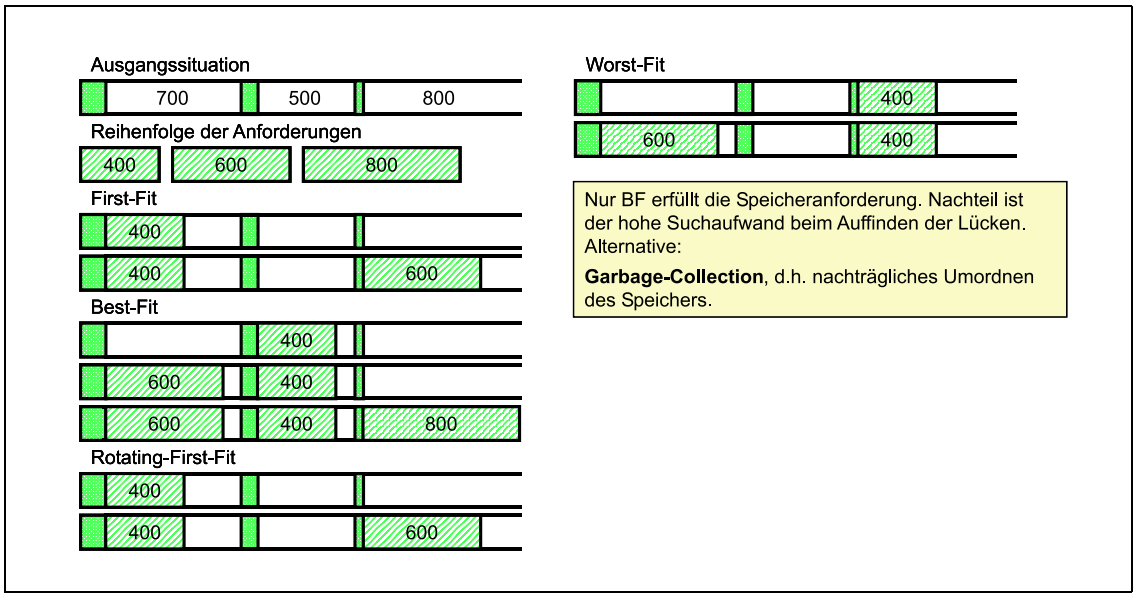


Abbildung 7.3: Unbrauchbarkeit von First-Fit

Collection). Durch die Segmentierung kann die Situation eintreten, dass die größte Lücke nicht mehr ausreicht, um ein Segment zu speichern, jedoch der Platz in der Summe der Lücken vorhanden ist. Dann kann entweder das Segment auf mehrere Lücken verteilt werden oder man schiebt die Segmente zusammen. Muss allerdings häufiger zusammengeschoben werden, kann man davon ausgehen, dass der Speicher überlastet ist. Dann ist es besser, einen Teil der Prozesse abzubrechen.

7.3 Buddy-Systeme

Eine andere Speicheraufteilung bilden die so genannten Buddy-Systeme. Die grundlegende Idee der **Buddy-Systeme** – Buddy kann sinngemäß mit Kumpel übersetzt werden – ist es, eine Organisationsform bereitzustellen, welche im Mittel weniger Verschnitt bzgl. der Lückenstruktur der Segmente hat und auch weniger starr ist als z.B. das Paging-Konzept.

7.3.1 Einfache Buddy-Systeme

Bei einem Buddy-System lässt man nur Anforderungen in Größe einer 2er-Potenz zu. Sei die Gesamtgröße des physikalischen Speichers 2^{max}. Bei einer Anforderung teilt man den Speicher solange in kleinere 2er-Potenzen auf, bis die gewünschte Größe verfügbar ist. Für jede 2er-Potenz gibt es eine Liste L_{pot}, welche die momentan freien Speicherbereiche bzgl. der Größe 2^{pot} angibt.

Die beiden wesentlichen Aufgaben des Buddy-Systems sind die Platzierung von Segmenten sowie die Freigabe von nicht mehr benötigtem Speicherplatz. Bei der Platzierung eines Segments der Größe 2^{pot} wird in der entsprechenden Liste L_{pot} geprüft, ob dort freier Speicherplatz vorhanden ist. Folgende Fälle sind zu unterscheiden:

$L_{pot} \neq \emptyset$**:** Platziere Segment und lösche entsprechenden Speicherplatz aus der Liste.

$L_{pot} = \emptyset$**:** Überprüfe die nächsthöhere Liste L_{pot+1}, dann L_{pot+2} usw. so lange, bis eine nichtleere Liste gefunden wurde. Zerlege einen dieser Listenkandidaten solange

in zwei Hälften (Buddies), bis eine passende Buddygröße entsteht. Dort wird dann das Segment abgelegt. Sind alle Listen bis L_{max} leer, so ist eine Platzierung des Segments unmöglich.

Abbildung 7.4 stellt ein Beispiel für Buddy-Systeme dar. Sei $max = 18$. Es werde eine Anforderung der Größe 2^{16} betrachtet.

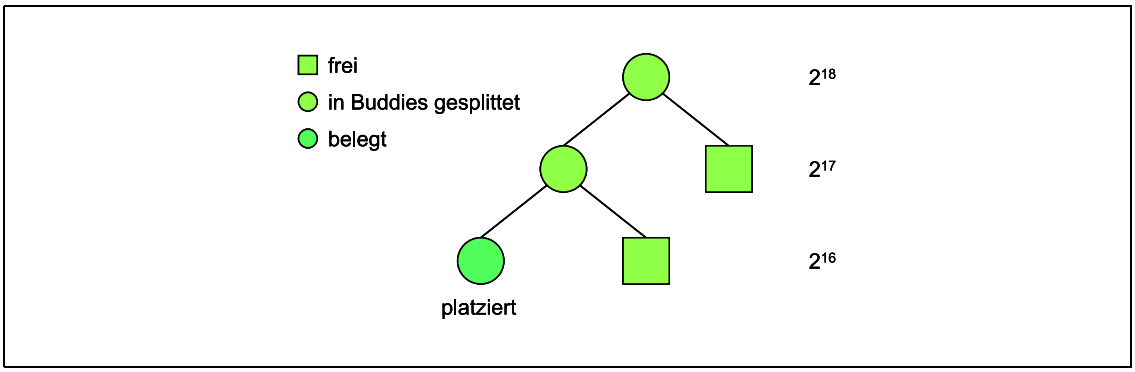


Abbildung 7.4: Beispiel eines Buddy-Systems

Bei der Freigabe eines Segments der Größe 2^{pot} ist nachzusehen, ob der zugehörige Buddy (mit gemeinsamer Wurzel) ebenfalls frei ist. Falls ja, so vereinige beide wieder zu einem Bereich der Größe 2^{pot+1}. Iteriere dies solange, bis keine Buddies mehr vereinigt werden können.

Die Anfangsadressen der jeweiligen Buddies bzw. die Adresse des wiedervereinigten Blocks berechnen sich äußerst einfach. Angenommen, es gäbe einen Speicherbereich $[0 : 2^{max} - 1]$ (binär $[0 \ldots 0 : 1 \ldots 1]$). Ein Speicherbereich wird durch ein 2-Tupel bestehend aus Anfangsadresse und Länge gekennzeichnet. Ein Speicherbereich der Größe 2^k hat folgendes Aussehen: $[\alpha 0 \ldots 0 : \alpha 1 \ldots 1]$, wobei die Anzahl der Nullen bzw. Einsen k beträgt. Der Präfix α charakterisiert den Speicherbereich, siehe Abbildung 7.5.

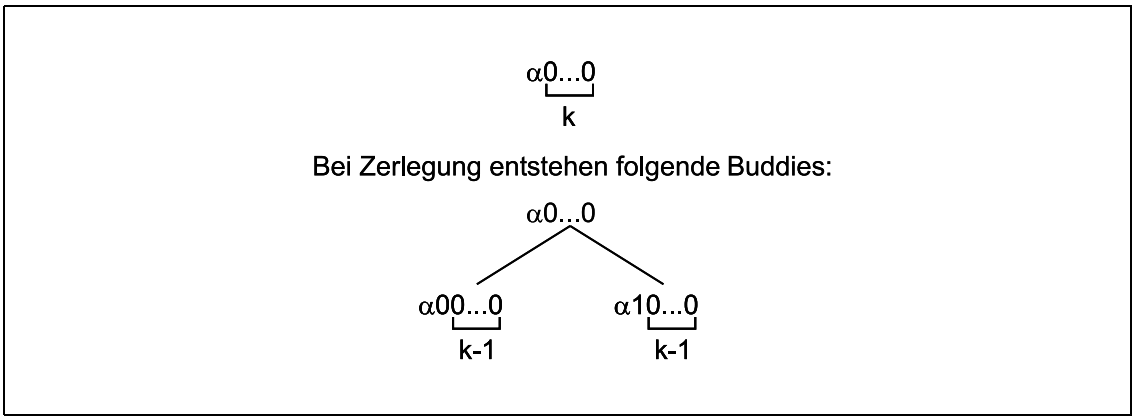


Abbildung 7.5: Zerlegung von Buddies

Beide neu entstandenen Buddies unterscheiden sich nur in einem Bit. Das Verfahren erzeugt eine baumartige Struktur (die Zahlen an den Knoten entsprechen jeweils der Vorsilbe α). Es kann vorkommen, dass benachbarte Speicherbereiche frei, jedoch keine Buddies sind. Dann lassen sich diese Bereiche nicht zu einem großen Bereich zusammenfassen. Ein Beispiel eines Buddy-Systems unter Angabe entsprechender Listen ist in Abbildung 7.6 zu sehen.

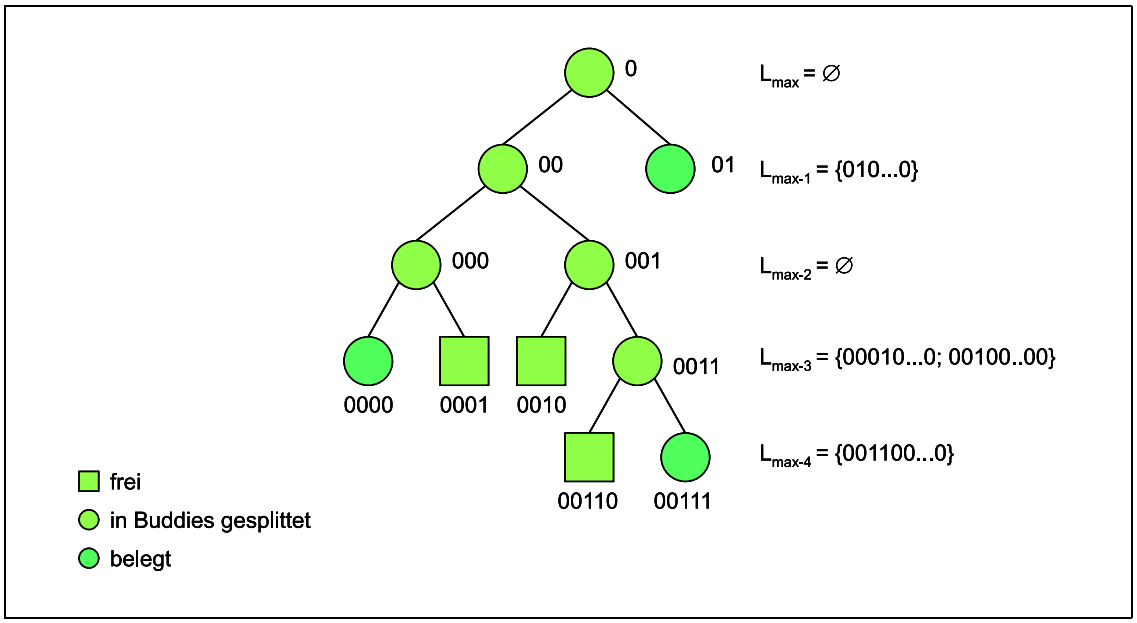


Abbildung 7.6: Listenangabe in einem Buddy-System

7.3.2 Gewichtete Buddy-Systeme

Um eine feinere Aufteilung des Speichers zu erreichen, können **gewichtete Buddies** verwendet werden. Dabei ist ein Block der Größe 2^{r+2} z.B. im Verhältnis $1:3$ in Blöcke der Größe 2^r bzw. $3 \cdot 2^r$ zu zerlegen. Ein Buddy der Größe $3 \cdot 2^r$ wird im Verhältnis $2:1$ in Buddies der Größen 2^{r+1} und 2^r zerlegt. Der Vorteil des Verfahrens liegt in der mehrstufigen Aufteilung der einzelnen Buddy-Größen. Nachteilig ist der erhöhte Aufwand. Insbesondere ist die Adressberechnung wesentlich komplizierter.

Beispielhaft wird in Abbildung 7.7 die Aufteilung eines Speicherbereichs der Größe 2^4 mit gewichteten Buddies gezeigt.

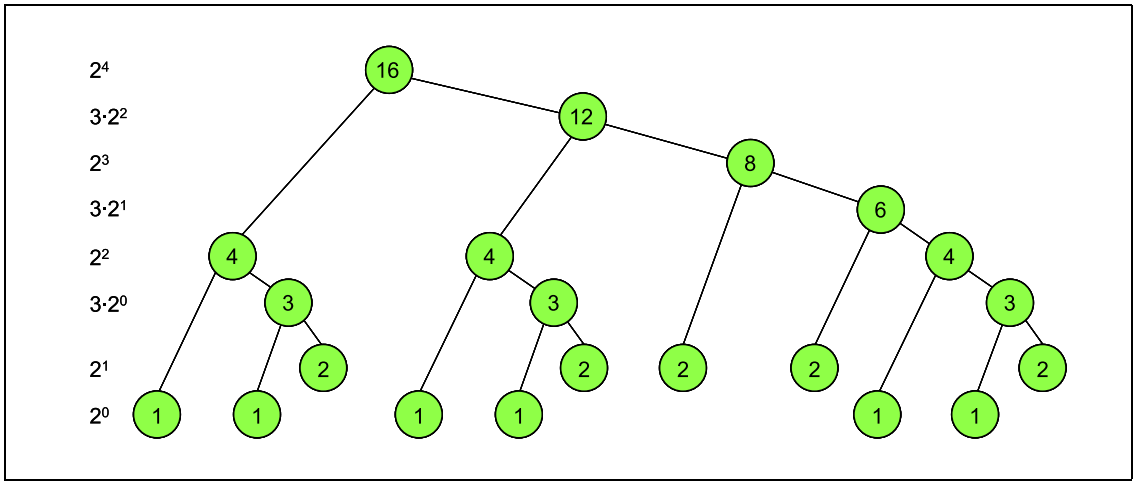


Abbildung 7.7: Speicherbereich mit gewichteten Buddies

Die Vorgehensweise bei der Zuweisung von Blöcken in gewichteten Buddy-Systemen ist in Abbildung 7.8 graphisch dargestellt. Der Algorithmus gibt an, wie die Zuweisung eines Blocks der Größe u mit Hilfe eines gewichteten Buddy-Systems vorgenommen wird. Sei dabei $u \in \{2^0, 2^1, \ldots, 2^m\} \cup \{3 \cdot 2^0, \ldots, 3 \cdot 2^{m-2}\}$, und seien $L_1, L_2, \ldots, L_{2m}$ die Listen der

zulässigen Blockgrößen, wobei einer hohen Blockgröße ein hoher Listenindex entspricht.

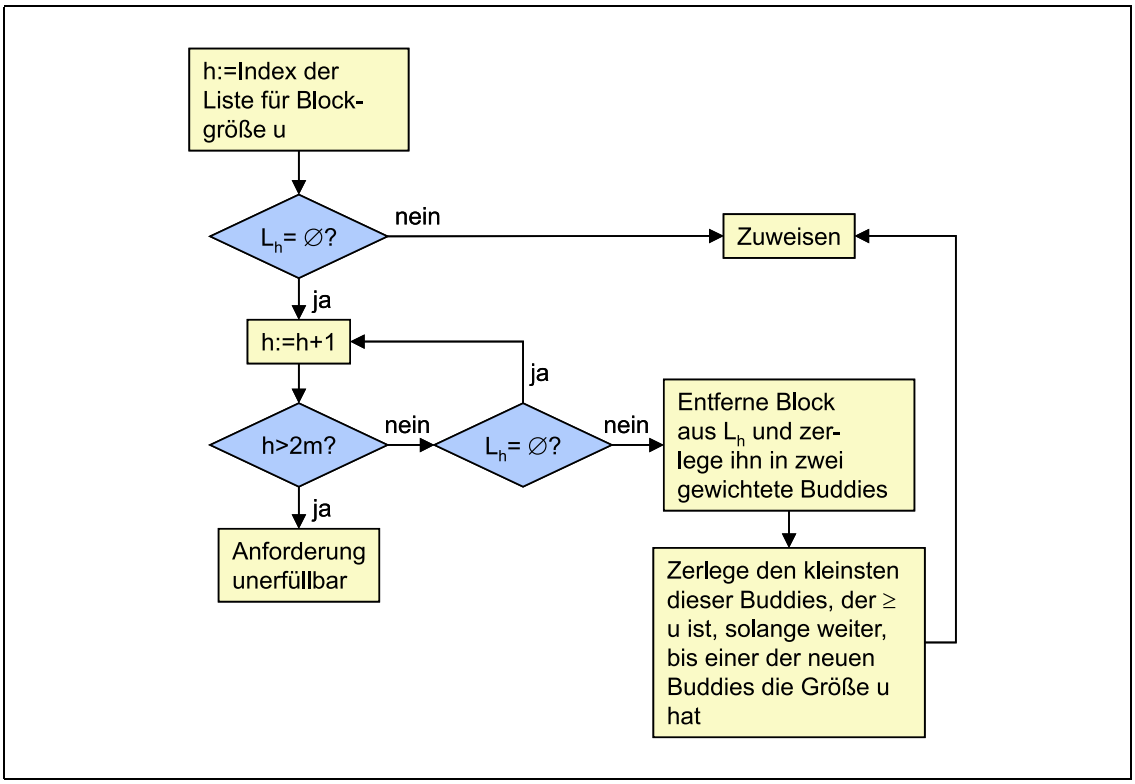


Abbildung 7.8: Funktionsweise von gewichteten Buddy-Systemen

7.4 Paging

Das Paging hat unter den heutigen Speicherverwaltungsmethoden einen so hohen Stellenwert, dass wir ihm einen eigenen Abschnitt widmen wollen. Im Gegensatz zu Segmentierung und Buddy-Systemen werden beim **Paging** ausschließlich Speicherblöcke einheitlicher Größe verwendet; man bezeichnet sie als Seiten oder **Pages**. Dies bedeutet zunächst, dass der logische Adressraum von Programmen, die auf einem Hintergrundspeicher liegen, in Seiten unterteilt wird. Analog dazu erfolgt eine Aufteilung des physikalischen Adressraums (Hauptspeicher) in Rahmen (**Frames**), die jeweils genau eine Seite aufnehmen können. Zur Ausführung des Programms werden dann alle (oder auch nur einige) der im Hintergrund gespeicherten Seiten in die Rahmen des Hauptspeichers geladen.

Der Vorteil dieses Verfahren liegt in der Vermeidung der Speicherzerstückelung (**externe Fragmentierung**). Als Nachteil muss man hingegen einen eventuellen Verschnitt innerhalb einer Seite hinnehmen, da die Größe eines Programmoduls im Allgemeinen kein ganzzahliges Vielfaches einer Seite ist (**interne Fragmentierung**).

Es stellt sich die Frage, wann eine Seite aus dem Hintergrund in den Hauptspeicher geladen werden soll. Die folgenden Strategien kommen in Betracht:

Demand-Paging: Fehlt im Hauptspeicher eine Seite, d.h. es liegt ein **Seitenfehler** vor, so wird die fehlende Seite nachgeladen. Als Problem verbleibt, welche Seite dafür verdrängt werden soll (Replacement-Strategy).

Demand-Prepaging: Arbeitet wie Demand-Paging, jedoch werden im Fall eines Seitenfehlers gleich mehrere Seiten geladen und verdrängt.

Look-Ahead: Nachladeoperationen werden bei dieser Strategie auch durchgeführt, ohne dass unbedingt ein Seitenfehler vorliegt.

Bezüglich der Anzahl der Seitenfehler ist Demand-Paging optimal, da diese Strategie die Zahl der Seitentransporte minimiert. Betrachtet man jedoch andere Kostenmaße, so kann u.U. auch Demand-Prepaging oder Look-Ahead optimal sein (12 Seiten auf einmal nachzuladen, kann billiger sein, als das Nachladen von 12 Seiten hintereinander).

7.4.1 Demand-Paging-Strategien

Im Laufe der Abarbeitung eines Programms werden immer wieder Zugriffe auf den Hauptspeicher notwendig. Die Frage, die nun behandelt werden soll, lautet, wie man (bei bekanntem Programmablauf) eine besonders hohe Trefferquote (**Hit-Ratio**) bzgl. der im Hauptspeicher vorhandenen Seiten erreichen kann. Anders ausgedrückt geht es darum, beim Nachladen die jeweils unwichtigsten Seiten zu verdrängen. Hierzu betrachten wir

- den logischen Adressraum N mit n Seiten: $N = \{0, \ldots, n-1\}$, sowie
- den physikalischen Adressraum M mit m Seitenrahmen: $M = \{0, \ldots, m-1\}$.

Im Allgemeinen gilt dabei $n \gg m$, d.h. der logische Adressraum ist i.d.R. wesentlich größer als der physikalische. Die Ausführung eines Programms wird durch seine Folge von Seitenzugriffen charakterisiert. Diese Folge nennen wir **Referenzstring** und bezeichnen ihn mit ω.

Sei $\omega = r_1 r_2 \ldots r_T \in N^T$ der Referenzstring einer Programmausführung. r_1 bezeichnet den ersten und r_T den letzten Seitenzugriff. Wird auf eine Seite mehrmals hintereinander zugegriffen, so sind entsprechend mehrere aufeinanderfolgende r_i identisch.

Sei weiter $S_t := \{i | i \in N \wedge i \text{ belegt Seitenrahmen in } M\}$ der Speicherzustand, d.h. die Menge der aktuell im Hauptspeicher befindlichen Seiten.

Dann lässt sich ein **Seitenaustauschalgorithmus** A als endlicher Automat ohne Ausgabe folgendermaßen beschreiben:

DEFINITION

Seitenaustauschalgorithmus

$$
\begin{aligned}
A &= (N, \{S_t\} \times Q, q_0, g_A) \quad \text{mit} \\
N &= \text{Eingabemenge,} \\
Q &= \text{Menge der Kontrollzustände,} \\
\{S_t\} \times Q &= \text{Speicherzustand und Kontrollzustand,} \\
q_0 &= \text{Startzustand und} \\
g_A &= \text{Überführungsfunktion.}
\end{aligned}
$$

Die Überführungsfunktion lautet formal:

$$
\begin{aligned}
g_A : N \times (\{S_t\} \times Q) &\to \{S_t\} \times Q \\
(r_i, S, q) &\mapsto (S', q')
\end{aligned}
$$

D.h. wenn r_i die nächste benötigte Seite und der Systemzustand (S, q) ist, dann gehe zu einem neuen Systemzustand (s', q') über, der eventuell einen Seitenaustausch beinhaltet.

Die verschiedenen Demand-Paging-Strategien unterscheiden sich durch die unterschiedliche Wahl der im Bedarfsfall zu ersetzenden Seite (Ersetzungsstrategie). Wir wollen im Folgenden einige solcher Strategien näher betrachten.

First-In-First-Out

Die **FIFO**-Strategie ordnet die Seiten im Hauptspeicher nach ihrem Alter an. Im Bedarfsfall wird die älteste Seite verdrängt.

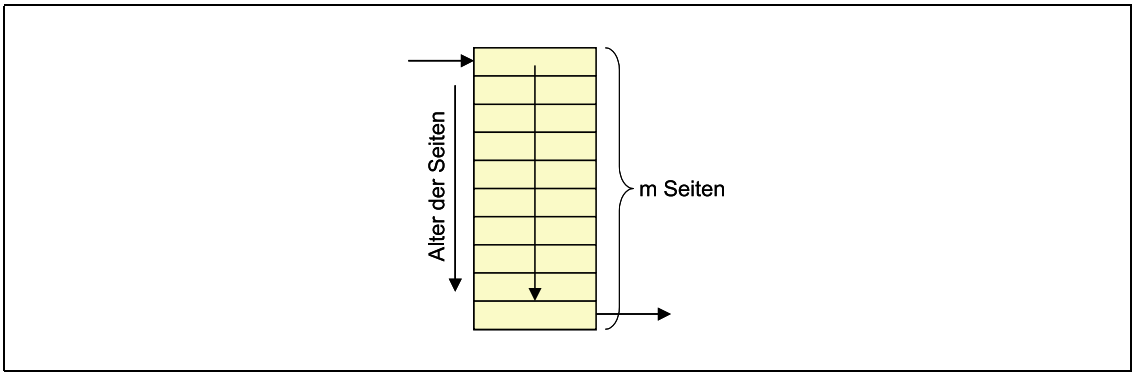


Abbildung 7.9: Funktionsweise von FIFO

Angenommen, der Hauptspeicher fasst m Seitenrahmen. Sei der Kontrollzustand $q = (q_1, \ldots, q_m)$ mit $q_i \in \{0, \ldots, n-1\}$, wobei q_1 die Nummer der jüngsten und q_m die Nummer der ältesten Seite im Hauptspeicher ist. Die Übergangsfunktion g_{FIFO} für die FIFO-Strategie sieht dann wie folgt aus:

$$\begin{aligned} g_{FIFO} : N \times (\{S_t\} \times Q) &\to \{S_t\} \times Q \\ (r_i, S, q) &\mapsto (S', q') \end{aligned}$$

derart, dass

- $S' = S$, $q' = q$, falls $r_i \in S$ (kein Seitenfehler)
- $S' = S \cup \{r_i\}$, $q' = (r_i, q_1, \ldots, q_k)$, falls $r_i \notin S \wedge \mid S \mid < m$ und $S = (q_1, \ldots, q_k)$
- $S' = S \cup \{r_i\} \setminus \{q_m\}$, $q' = (r_i, q_1, \ldots, q_{m-1})$, falls $r_i \notin S \wedge \mid S \mid = m$

In Abbildung 7.10 ist die Funktionsweise von FIFO anhand eines Beispiels ersichtlich.

Least-Recently-Used

Die **LRU**-Strategie arbeitet wie die FIFO-Strategie, jedoch wird bei jedem Zugriff auf eine Seite, die bereits im Hauptspeicher ist, diese verjüngt. Im Bedarfsfall muss die Seite mit der maximalen **Rückwärtsdistanz**, also die am längsten nicht mehr benutzte Seite, ersetzt werden (siehe Abbildung 7.11). Realisierbar ist eine solche Strategie am einfachsten mit einer verketteten Liste, da ansonsten ein hoher Aufwand an Umspeicherungen entstehen kann.

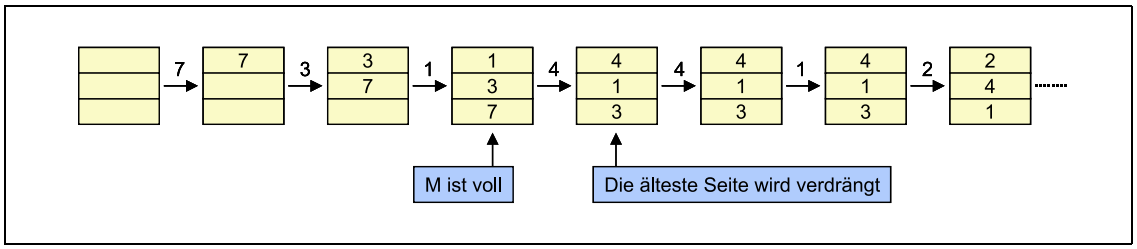


Abbildung 7.10: Beispiel für FIFO

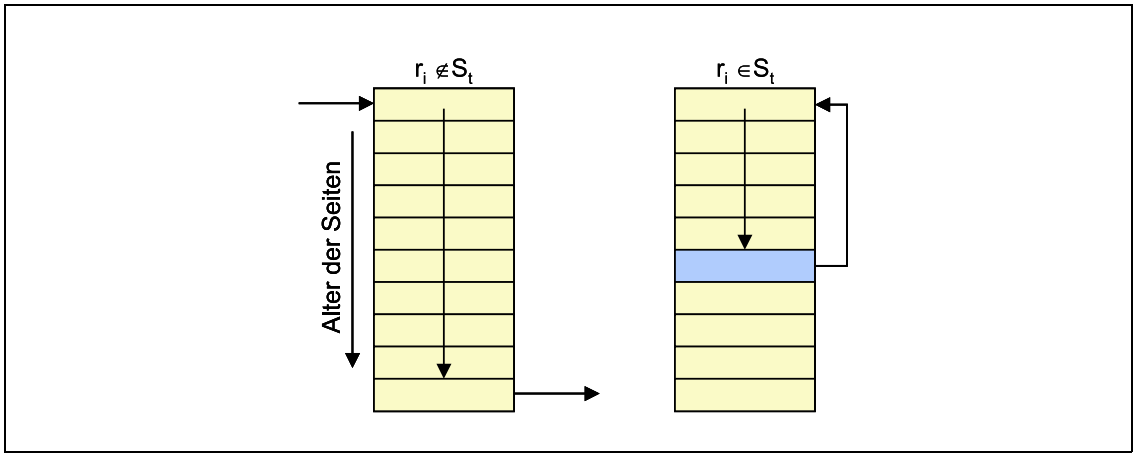


Abbildung 7.11: Funktionsweise von LRU

Second-Chance

Second-Chance stellt einen Kompromiss zwischen FIFO und LRU dar. Prinzipiell arbeitet die Strategie wie FIFO, jedoch existiert ein zusätzliches Bit (**Use-Bit**) für jede Seite, welches bei einer nochmaligen Benutzung der Seite gesetzt wird (siehe Abbildung 7.12). Ist eine neue Seite zu laden, so verdrängt die Strategie die älteste Seite mit nicht gesetztem Use-Bit. Die Use-Bits sind zu löschen, wenn

- alle Bits gesetzt sind, da dann daraus keine Informationen mehr zu gewinnen sind, und
- beim Eintreten eines Seitenfehlers.

Mit einem zusätzlichen Bit (**Modify-Bit**) kann diese Strategie noch verfeinert werden. Das Modify-Bit wird gesetzt, falls eine Seite im Hauptspeicher geändert wurde. Dies impliziert, dass die Seite nicht nur verdrängt, sondern auch zurück in den Hintergrundspeicher geschrieben werden muss. Dadurch verursacht das Verdrängen solcher Seiten höhere Kosten. Definiert man für jede Kombination (Use-Bit, Modify-Bit) eine Klasse, so sollte zuerst das älteste Element der Klasse (0, 0) verdrängt werden. Ist diese Klasse leer, so werden die Klassen (0, 1), (1, 0) und (1, 1) in dieser Reihenfolge untersucht, und die älteste Seite der ersten nichtleeren Klasse wird aus dem Speicher geworfen.

Ein Nachteil von FIFO und Second-Chance besteht darin, dass alte Seiten ausgelagert werden, auch wenn sie oft in Gebrauch sind. Andererseits sind diese beiden Strategien einfach zu realisieren. Weiter ist anzumerken, dass sich, mit Ausnahme von theoretischen Ansätzen, für jedes Verfahren ein Beispiel konstruieren lässt, bei dem es äußerst schlecht dasteht.

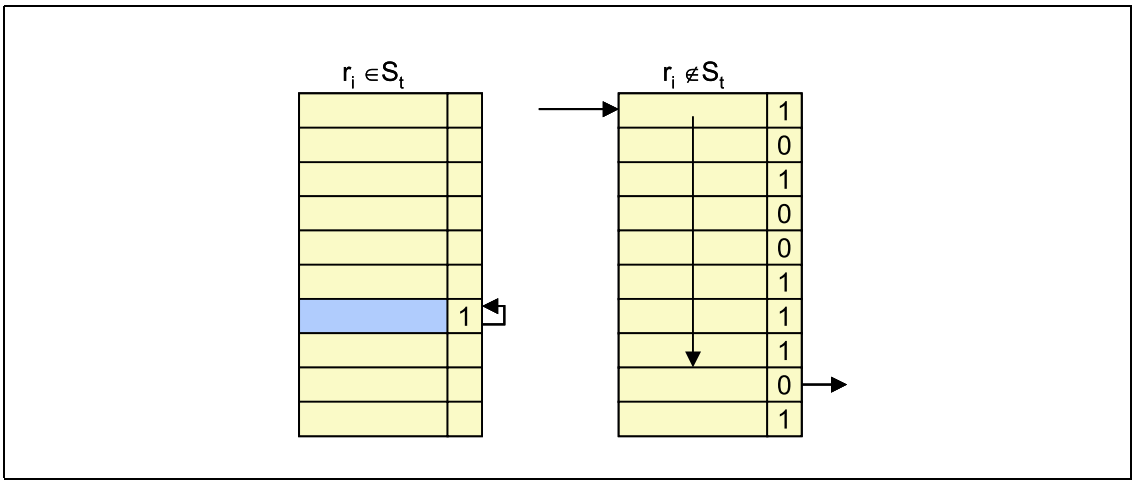


Abbildung 7.12: Funktionsweise von Second-Chance

Least-Frequently-Used

Die **LFU**-Strategie tauscht im Bedarfsfall die Seite mit der niedrigsten Nutzungshäufigkeit aus. Dabei gibt es mehrere Varianten, die Nutzungshäufigkeit zu messen, beispielsweise betrachtet man die Zugriffe auf die fragliche Seite

- seit Beginn des Referenzstrings,
- innerhalb der letzten h Zugriffe (Fenster der Größe h) oder
- seit dem letzten Seitenfehler.

Climb

Bei der **Climb**-Strategie steigt eine Seite bei jedem Aufruf eine Position höher, wenn sie bereits im Speicher vorhanden ist. Sie tauscht also ihre Position mit der vor ihr stehenden Seite. Ist die Seite nicht im Speicher vorhanden, so wird die Seite auf der untersten Position verdrängt und die neue Seite dorthin geladen (siehe Abbildung 7.13).

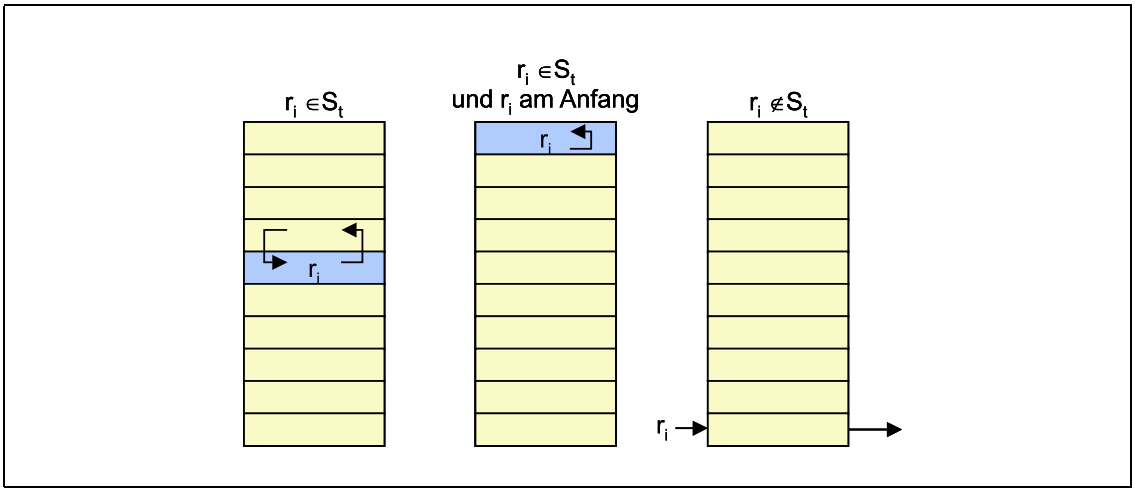


Abbildung 7.13: Funktionsweise von CLIMB

Random

Im Falle eines Seitenfehlers wird die Seite, die aus dem Hauptspeicher verdrängt wird, ausgewürfelt (also zufällig bestimmt).

OPT

Die optimale Strategie arbeitet nach dem Prinzip, die Seite mit dem größten **Vorwärtsabstand** auszutauschen. Die Seite also, die am längsten nicht mehr gebraucht werden wird, ist Tauschkandidat. OPT verursacht die wenigsten Seitenfehler unter allen Demand-Paging-Algorithmen, ist aber nur in Ausnahmefällen realisierbar. Man kann das Verfahren jedoch approximieren, indem man die Seite als Tauschkandidat wählt, deren erwarteter Vorwärtsabstand am höchsten ist. Dieser Erwartungswert kann z.B. aufgrund von Ergebnissen des zurückliegenden Teils des Referenzstrings geschätzt werden. OPT eignet sich sehr gut zur Bewertung der Güte eines Verfahrens (Strategie X ist max. x% schlechter als OPT).

7.4.2 Nicht-Demand-Paging-Strategien

Nicht-Demand-Paging-Algorithmen nehmen bei Bedarf mehrere (und nicht nur erzwungene) Seitentransporte vor. Die Strategien nutzen die Tatsache aus, dass Programme meist geographisch lokal arbeiten: Wenn auf eine Seite r_i zugegriffen wird, ist die Chance, dass die Nachbarn r_{i+1} (rechter Nachbar) und r_{i-1} (linker Nachbar) ebenfalls referenziert werden, sehr groß. Gute Beispiele hierfür sind sequentielle Schleifendurchläufe oder die Ausführung von einfachem Programmcode.

Unter den Nicht-Demand-Paging-Algorithmen ist der OBL-Algorithmus (One-Block-Look-Ahead) am bekanntesten. OBL arbeitet, solange kein Seitenfehler auftritt, im Wesentlichen wie LRU. Es existieren mehrere Varianten dieser Strategie, von denen zwei hier vorgestellt werden sollen.

OBL als Demand-Prepaging-Version

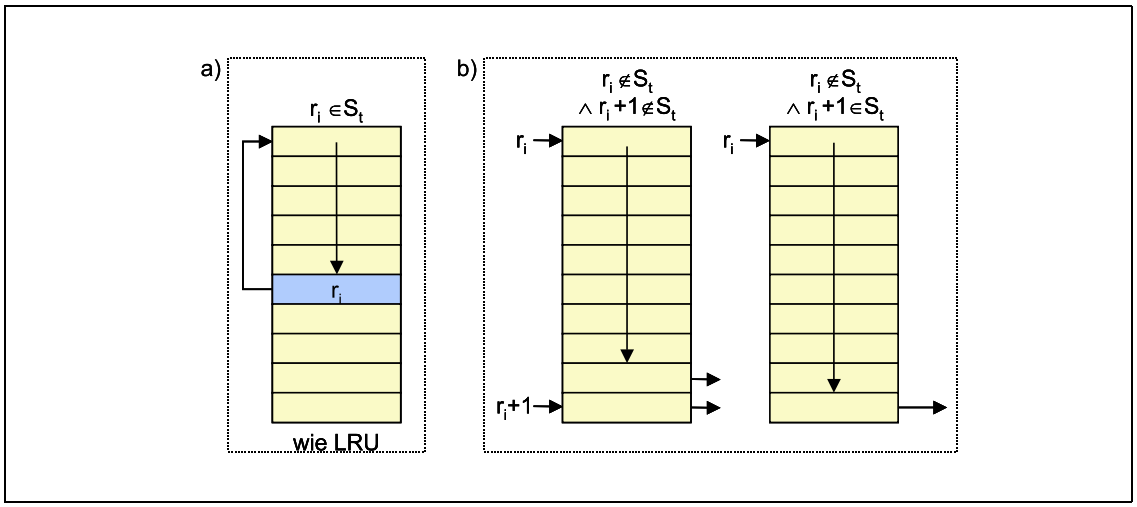


Abbildung 7.14: Funktionsweise von OBL als Demand-Prepaging-Version

Falls r_i bereits im Speicher vorhanden ist, verhält sich das Verfahren analog zu LRU (siehe Abbildung 7.14). Unterschiede treten auf, wenn aufgrund eines Seitenfehlers r_i nachzuladen ist. In diesem Fall wird unterschieden, ob die auf r_i folgende Seite r_{i+1} bereits im Speicher vorhanden ist oder nicht. Ist sie nicht vorhanden, so ist sie an die letzte Stelle im Speicher zu laden. Ist die Seite r_{i+1} allerdings vorhanden, so ist die Vorgehensweise wie bei FIFO.

OBL als Look-Ahead-Variante

Die Look-Ahead-Variante ist analog zur Demand-Prepaging-Version bis auf den Fall, dass die Seite r_i bereits im Speicher ist (siehe Abbildung 7.15). Dann ist zu unterscheiden, ob die nachfolgende Seite r_{i+1} bereits im Speicher ist oder nicht. Falls nicht, so wird sie zusätzlich an die letzte Position geladen. Ist sie bereits vorhanden, so verhält sich die Variante wie die LRU-Strategie.

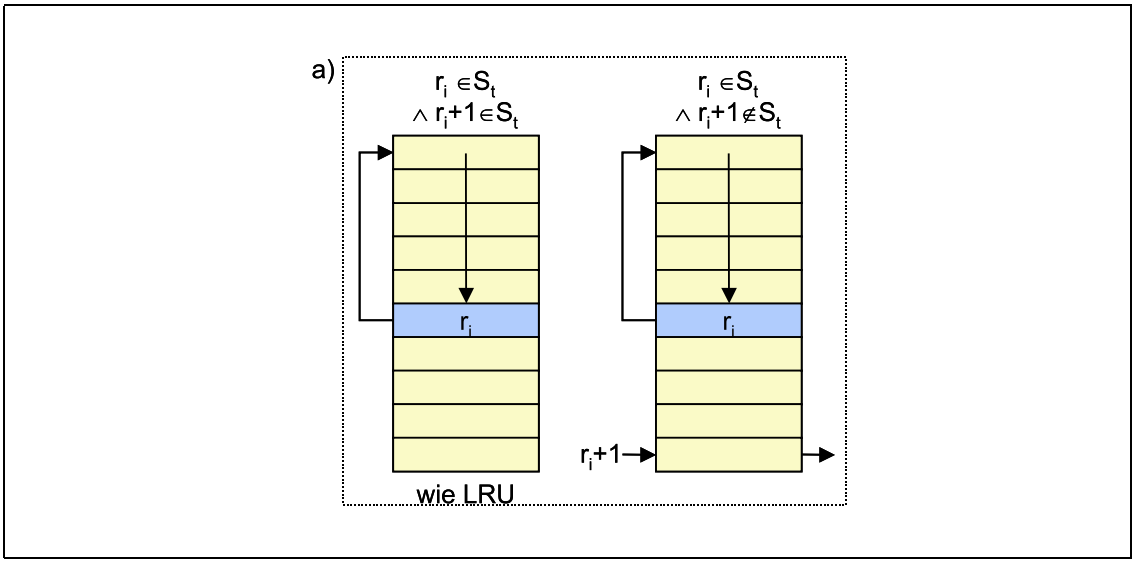


Abbildung 7.15: Funktionsweise von OBL als Look-Ahead-Version

7.4.3 Diskussion der Paging-Algorithmen

An dieser Stelle sollen beide Arten des Pagings gegenübergestellt werden. Als Zielgröße betrachten wir dabei im Folgenden die Kosten.

Kosten von Paging-Algorithmen

Eine Möglichkeit der Bewertung von Algorithmen sind ihre Kosten. Unter **Kosten** versteht man den Aufwand für die Speicherzustandsänderung.

Im weiteren Verlauf nehmen wir an, dass die Verdrängungskosten proportional zu den Ladekosten sind. Ferner sind alle Kosten für das Laden und ggf. das Verdrängen auf eins zu normieren.

Sei S_t der Speicherzustand zum Zeitpunkt t, d.h. der Zustand nach dem t-ten Speicherzugriff. S_t berechnet sich zu

$$S_t = S_{t-1} \cup X_t \setminus Y_t.$$

Dabei bezeichnet X_t die Menge der neu geladenen und Y_t die Menge der verdrängten Seiten.

Bei Anwendung eines Algorithmus A auf den Referenzstring $\omega = r_1 r_2 \dots r_T$ und bei Verwendung von m Hauptspeicherseiten entstehen folgende Kosten:

1. Gesamtzahl der Seitentransporte (Seitenfehler)

$$\alpha(A, m, \omega) := \sum_{t=1}^{T} |X_t|$$

2. Kosten für das Laden von i Seiten =: $h(i)$ mit

$h : N_0 \to N_0$ mit $h(0) := 0$
$h(1) := 1$ (normiert)
$h(i) \geq h(i-1)$

3. Gesamtkosten C für den Referenzstring ω :

$$C(A, m, \omega) := \sum_{t=1}^{T} h(|X_t|)$$

Wie man sich leicht überlegen kann, gibt es für jeden Nicht-Demand-Paging-Algorithmus einen Demand-Paging-Algorithmus, der bei beliebigem Referenzstring höchstens ebensoviele Seitenfehler macht wie besagter Nicht-Demand-Paging-Algorithmus.

Dass Nicht-Demand-Paging-Algorithmen dennoch von Bedeutung sind, lässt sich damit begründen, dass die Gesamtkosten bei einem Nicht-Demand-Paging-Algorithmus in Abhängigkeit vom Referenzstring ω und der Speichergröße m geringer sein können als beim entsprechenden Demand-Paging-Algorithmus.

SATZ

Zu jedem Algorithmus A existiert ein DPA (Demand-Paging-Algorithmus) A^* mit:

$$\alpha(A^*, m, \omega) \leq \alpha(A, m, \omega) \; \forall m, \omega$$

d.h. DPA ist optimal bezüglich der Zahl der Seitentransporte.

Die Aussage, dass zu jedem Algorithmus A ein DPA A^* existiert mit:

$$C(A^*, m, \omega) \leq C(A, m, \omega) \; \forall m, \omega,$$

(d.h. DPA ist optimal bezüglich der Kosten) gilt jedoch nur, wenn das gemeinsame Nachladen von k Seiten nicht billiger ist als das einzelne Laden von k Seiten (also $h(k) \geq k \cdot h(1) = k$ für $k \geq 0$).

Die folgenden Betrachtungen beziehen sich auf Demand-Paging-Algorithmen. Wir stellen die Frage, wie sich die Seitenfehlerzahl und die Gesamtkosten für einen Referenzstring ω in Abhängigkeit von der Seitenzahl m verändern.

Zunächst liegt die Vermutung nahe, dass sich die Zahl der Seitenfehler (und damit bei DPAs die Kosten) mit wachsendem m verringert. Auch die Trefferquote (Hit-Ratio) sollte

sich verbessern. Jedoch ist diese Vermutung für einige Strategien und bei bestimmten pathologischen Referenzstrings überraschenderweise falsch. Die Vermutung gilt z.B. nicht für FIFO, Second-Chance und Climb, ist aber für LRU, OPT und LIFO gültig. Dieses Phänomen soll am Beispiel von FIFO veranschaulicht werden.

FIFO-Anomalie

Es gibt m, ω mit: $C(FIFO, m+1, \omega) > C(FIFO, m, \omega)$
$\alpha(FIFO, m+1, \omega) > \alpha(FIFO, m, \omega)$

BEISPIEL

FIFO-Anomalie

Sei $m = 3; \omega_1 = 2\ 3\ 0\ 1; \omega_2 = 2\ 0\ 3\ 1\ 4\ 2\ 5\ 3\ 0\ 4\ 1\ 5; \omega = \omega_1 \cdot \omega_2^k$

ω	2	3	0	1	2	0	3	1	4	2	5	3	0	4	1	5	2	0	3	1	4	2	5	3	...
3 Seiten	2	3	0	1	2		3		4		5		0		1		2		3		4	...			
		2	3	0	1		2		3		4		5		0		1		2		3	...			
			2	3	0		1		2		3		4		5		0		1		2	...			
Seitenfehler					x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
4 Seiten	2	3	0	1					4	2	5	3	0	4	1	5	2	0	3	1	4	2	5	...	
		2	3	0					1	4	2	5	3	0	4	1	5	2	0	3	1	4	2	...	
			2	3					0	1	4	2	5	3	0	4	1	5	2	0	3	1	4	...	
				2					3	0	1	4	2	5	3	0	4	1	5	2	0	3	1	...	
Seitenfehler									x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	...	

Daraus kann für dieses Beispiel gefolgert werden:

$$\frac{C(FIFO, 3, \omega = \omega_1 \cdot \omega_2^k)}{C(FIFO, 4, \omega = \omega_1 \cdot \omega_2^k)} \overset{(k \to \infty)}{\longrightarrow} \frac{1}{2}$$

Die Vergrößerung des Speichers führt bei diesem speziellen ω zu einer Verdopplung der Seitenfehler und der Kosten. Der Grund für diese so genannte **FIFO-Anomalie** liegt darin, dass bei FIFO die Seiten unabhängig von ihrer Aktualität altern.

Interessant ist nun die Frage, für welche Algorithmen keine Beispiele der obigen Art konstruiert werden können. Die Antwort darauf führt uns zum Begriff des Stack-Algorithmus.

Stack-Algorithmen

Sei $S(A, m, \omega) = S(m, \omega)$ die Menge der Seitennummern, welche am Ende der Abarbeitung eines Referenzstrings ω durch einen DPA unter Verwendung von m Rahmen im Speicher stehen.

DEFINITION

Stack-Algorithmus

Ein Algorithmus heißt **Stack-Algorithmus** genau dann, wenn

$$S(m, \omega) \subseteq S(m+1, \omega) \quad \forall\, m, \omega$$

erfüllt ist.

Daraus kann man folgern, dass für jeden Stack-Algorithmus gilt:

- Mit wachsender Speichergröße sinkt die Fehlerrate. Ebenso sinken die Nachladekosten.
- Als Konsequenz daraus ergibt sich, dass FIFO kein Stack-Algorithmus ist (ebenso wie Climb und einige andere mehr).

LRU, LIFO und OPT sind Stack-Algorithmen. Bei der Strategie LRU besteht dabei der Stack aus der Menge der zuletzt benötigten Seiten.

Prioritätsalgorithmen

DEFINITION

Prioritätsalgorithmus

Ein Demand-Paging-Algorithmus heißt **Prioritätsalgorithmus** genau dann, wenn es für alle ω (und unabhängig von m) eine Folge $\pi_1, \pi_2, \ldots, \pi_{T-1}$ von so genannten Prioritätslisten gibt, für die gilt:

1. π_i ist eine geordnete Liste der in $r_1, r_2, \ldots, r_i$ vorkommenden Seitennummern.
2. Ist $\omega = r_1 r_2 \ldots r_t$ und $|S(m,\omega)| = m$ und $r_{t+1} \notin S(m,\omega)$ (d.h. es muss eine Seite ersetzt werden), dann bestimmt sich die zu verdrängende Seite durch die Liste π_t wie folgt:

$$S(m, \omega, r_{t+1}) = S(m,\omega) \cup \{r_{t+1}\} \setminus \min_{\pi_t} S(m,\omega)$$

Dabei ist das Element niedrigster Priorität in $S(m,\omega)$ bzgl. der durch π_t gegebenen Anordnung.

Kurz gefasst heißt dies, dass Prioritätsalgorithmen das Austauschverhalten unabhängig von m durch Prioritätslisten bestimmen.

Die folgende Tabelle gibt Beispiele für Prioritätsalgorithmen und die dabei verwendeten Prioritätslistenanordnungen.

Algorithmus	Prioritätslistenanordnung
LRU	wachsende Rückwärtsdistanz
OPT	wachsende Vorwärtsdistanz
LIFO	wachsende Zeit des Eintritts in den Hauptspeicher
LFU	abnehmende Häufigkeit der Benutzung

Es gilt folgender Satz:

SATZ

1. A ist Prioritätsalgorithmus $\Rightarrow$ A ist Stack-Algorithmus
2. Zu jedem Stack-Algorithmus A existiert eine Folge von Prioritätslisten, d.h. A ist Stack-Algorithmus $\Rightarrow$ A ist Prioritätsalgorithmus

Auf einen Beweis wollen wir hier verzichten.

Fallstrick:
Warum ist FIFO kein Prioritätsalgorithmus? Man könnte doch beispielsweise als Prio-

ritätsliste die Ordnung der Seiten nach ihrem Alter im Hauptspeicher hernehmen und damit folgern:

$$\text{FIFO=Prioritätsalg.} \Rightarrow \text{FIFO=Stackalg.} \Rightarrow \text{Anomalie unmöglich}$$

im Widerspruch zum oben Gesagten. Der Argumentationsfehler hierbei liegt in der vorgeschlagenen Prioritätsliste, da diese von m abhängig ist!

KAPITEL 8

Speicherzuteilung bei Multiprogramming

Die bisher vorgestellten Ergebnisse beschränkten sich auf den Einprogrammbetrieb (siehe Abbildung 8.1). Zwar wird die Verwaltung im Mehrprogrammbetrieb (Multiprogramming) deutlich komplizierter, jedoch ist Multiprogramming wünschenswert, da der Einprogrammbetrieb aufgrund der Geschwindigkeitsdiskrepanz zwischen der CPU und den Endgeräten sehr ineffizient sein kann.

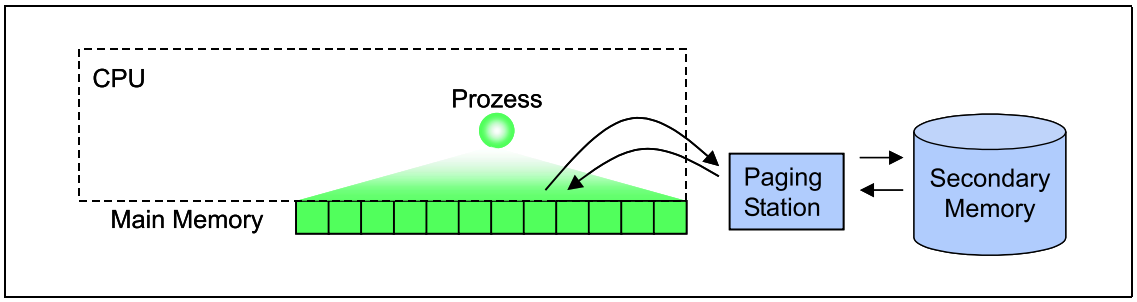


Abbildung 8.1: Bisherige Betrachtung: Speicherzuteilung im Einprogrammbetrieb

Beim Multiprogramming sind mehrere Programme gleichzeitig aktiv und greifen auf eine gemeinsame CPU zu. Wie soll man in diesem Fall den gemeinsamen Hauptspeicher für die verschiedenen Prozesse verwalten? Geht man beispielsweise von n gleichzeitig aktiven Prozessen aus (siehe Abbildung 8.2), so stellt sich die Frage, wie viele der insgesamt verfügbaren Rahmen jeder einzelne Prozess erhalten soll. Dass die naheliegende Idee, jedem Prozess gleichviele Rahmen (z.B. m Stück) zuzuteilen, sich nicht immer als optimal herausstellen wird, ist unmittelbar einsichtig: Manche Prozesse werden i.d.R. nicht alle zugeteilten Seiten brauchen, während andere mit ihrem Anteil hinten und vorne nicht auskommen.

Ein eng damit zusammenhängendes Problem (da n und m ja voneinander abhängen) ist die Frage nach dem optimalen Multiprogramminggrad n. Wählt man n zu klein, so verschwendet man offensichtlich Systemressourcen. Ein zu großes n führt dazu, dass die Zahl der Rahmen, die man einem einzelnen Prozess zuteilen kann, sinkt. Dadurch wächst die Zahl der Seitenfehler, bis im so genannten **Thrashing**-Bereich der Systemdurchsatz indiskutabel wird. Die schematische Abhängigkeit des Systemdurchsatzes D vom Multiprogramminggrad n ist Abbildung 8.3 dargestellt.

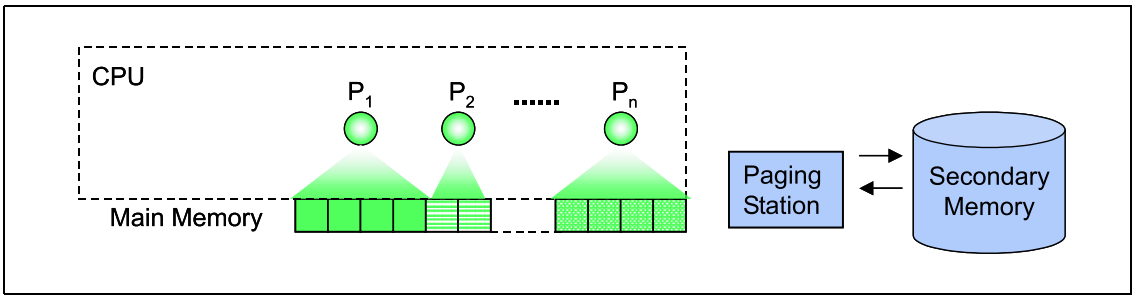


Abbildung 8.2: Speicherzuteilung bei Multiprogramming

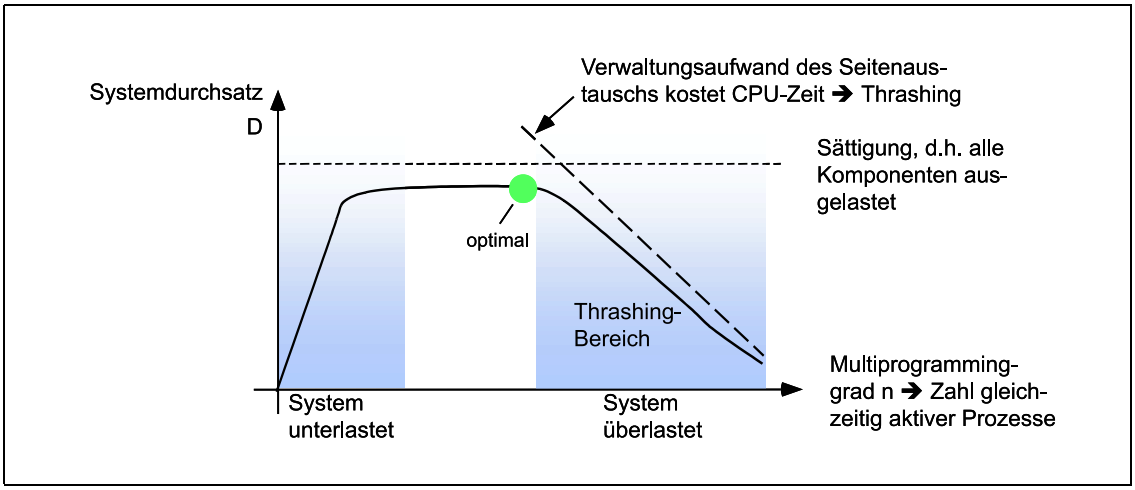


Abbildung 8.3: Thrashing durch steigenden Multiprogramminggrad

Aufteilung des Speichers auf aktive Programme

Wie soll man also sinnvollerweise den Speicher auf die zur Zeit aktiven Prozesse aufteilen? Wünschenswert ist sicherlich, die Größe der zugewiesenen Bereiche in Abhängigkeit von der Zeit festzulegen. Daneben ergeben sich eine Reihe von Detailfragen:

1. Wie viele Rahmen weist man einem Prozess zu?
2. Wann soll ein Prozess neu aufgenommen werden?
3. Wann ist ein aktiver Prozess stillzulegen (zu deaktivieren)?
4. Wann ändert sich der Speicherbereich?

Vorläufig können wir auf diese Fragen folgende allgemeine Antworten geben:

ad 1.: Man soll einem Programm so viele Rahmen geben, wie es braucht, also die Anzahl momentan aktiver Seiten. Dabei nennen wir eine Seite momentan aktiv, wenn die Wahrscheinlichkeit hoch ist, dass sie in naher Zukunft benötigt wird.

ad 2.: Ein neuer Prozess kann dann gestartet werden, wenn genügend Speicherplatz für seine aktiven Seiten verfügbar ist.

ad 3.: Ein Prozess ist dann stillzulegen, wenn ein Seitenfehler auftritt und alle Tauschkandidaten momentan aktiv sind.

ad 4.: Wenn die Zahl der aktiven Seiten eines Prozesses ansteigt und andere Prozesse Seiten belegt haben, die momentan nicht aktiv sind, so dürfen diese inaktiven Seiten von anderen Prozessen genutzt werden.

8.1 Die Lifetime-Funktion

Die **Lifetime-Funktion** $L(m)$ gibt die mittlere Zeit zwischen aufeinanderfolgenden Seitenfehlern in Abhängigkeit von der zugeordneten Rahmenzahl m an. Gewöhnlich (d.h. einige seltene Anomalien ausgenommen) steigt L mit wachsendem m monoton an. Je mehr Rahmen ein Prozess also im Hauptspeicher zur Verfügung hat, desto seltener werden die Seitenfehler, und desto mehr Zeit vergeht daher im Mittel zwischen zwei derselben. Die typische Gestalt einer Lifetime-Funktion hängt dabei sehr von den Prozessen ab, ist jedoch in den meisten Fällen sigmodal, d.h. die Form läßt sich näherungsweise als „S“ interpretieren (siehe Abbildung 8.4).

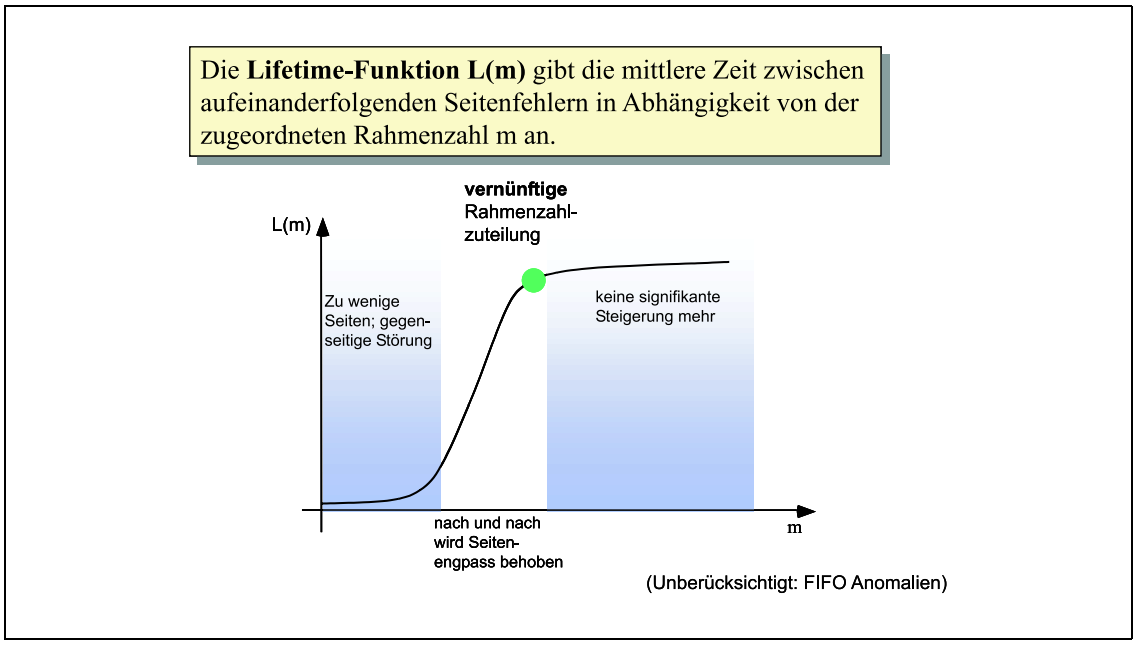


Abbildung 8.4: Lifetime-Funktion

Stehen einem Prozess nur wenige Rahmen zu, so kommt es häufig zu Seitenfehlern. Jeder weitere dem Prozess zugeteilte Rahmen bringt eine spürbare Verlängerung der Lifetime mit sich. Dies geht aber i.d.R. nicht ewig so weiter. Stattdessen mündet die Kurve in einen Sättigungsbereich, in dem weitere Rahmenzuteilungen kaum mehr Einfluss auf die Lifetime haben. Steht einem Prozess eine große Anzahl von Seiten zur Verfügung, kann es trotzdem zum Fehlen einer seltenen „exotischen” Seite kommen, was in einem erneuten Seitenfehler resultiert.

8.2 Das Working-Set und die Working-Set-Strategie

Zur Schätzung der Lifetime-Funktion ist der so genannte **Working-Set** von Bedeutung. Hierunter versteht man die Menge der in einem (noch präziser zu definierenden) Intervall der unmittelbaren Vergangenheit benötigten Seiten. Dieser Ansatz beruht auf folgenden

(empirisch motivierten) Annahmen:

- Die jüngere Vergangenheit ist eng mit der unmittelbaren Zukunft korreliert.
- Ein Prozess, der bislang viele aktive Seiten hatte, wird sich darin mit hoher Wahrscheinlichkeit nicht ändern.
- Über weite Bereiche verhalten sich Prozesse lokal: Seiten, die erst vor kurzem referenziert wurden, sind in der Zukunft wahrscheinlicher als solche, deren letzter Zugriff schon länger zurückliegt.

Diese Annahmen sind allerdings nicht allgemeingültig, Ausnahmen sind durchaus vorstellbar, beispielsweise dann, wenn ein Prozess in einen völlig neuen Abschnitt eintritt, also einen Phasenwechsel durchmacht.

Betrachten wir nun einen Prozess mit Referenzstring $\omega = r_1 r_2 r_3 \dots r_t \dots r_T$. Der Working Set $W(t,h)$ dieses Prozesses zur Zeit t unter einem Rückwärtsfenster der Größe h ist dann definiert als

$$W(t,h) := \bigcup_{i=t-h+1}^{t} r_i$$

$W(t,h)$ ist also die Menge der Seiten, die bei den letzten h Zugriffen mindestens einmal referenziert wurden (siehe Abbildung 8.5).

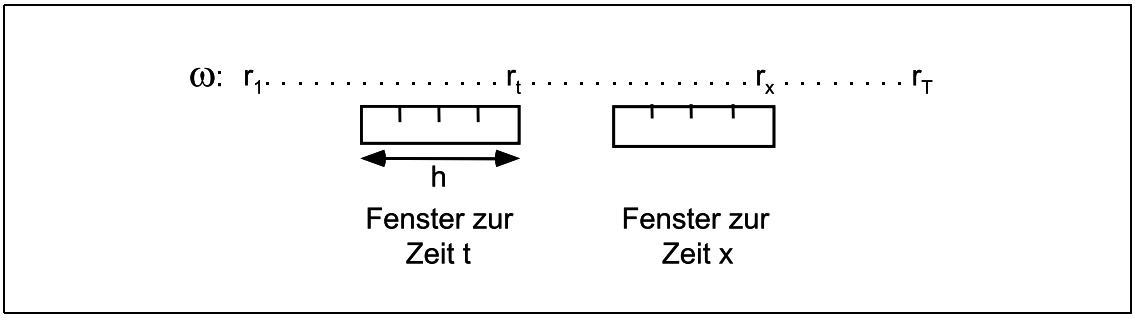


Abbildung 8.5: Working-Set

Sei $w(t,h)$ die Mächtigkeit von $W(t,h)$. Trivial ist, dass $w(t,h)$ mit wachsendem h steigt. Lokale Prozesse haben einen relativ kleinen Working Set, was bei nicht-lokalen Prozessen nicht der Fall ist. Problematisch ist nun die Wahl des Parameters h: Ist h zu klein gewählt, dann sind nicht alle aktiven Seiten im Working Set, ist h zu groß, dann befinden sich viele inaktive Seiten darin. Über die als nächstes erläuterte Working-Set-Strategie hat h direkten Einfluss auf die mittlere Anzahl m der zugeteilten Seiten pro Prozess und damit auch auf den Multiprogramminggrad n.

8.2.1 Die Working-Set-Strategie

1. h „gut“ einstellen.
2. Jedem Prozess so viele Rahmen zuteilen, wie seinem aktuellen Working-Set $W(t,h)$ entsprechen. Wird zusätzlicher Speicherplatz frei, so kann ggf. ein neuer Prozess aktiviert werden. Umgekehrt muss ein Prozess stillgelegt werden, wenn die Summe der Größen aller Working-Sets momentan zu hoch ist.

3. Bei Seitenfehlern sind solche Seiten Tauschkandidaten, die aktuell zu keinem Working-Set gehören. Der entsprechende Prozess „stiehlt“ sich eine dieser Seiten und gewinnt temporär eine neue Seite. Die Speicherzuteilung ist also variabel.

4. Ist kein Tauschkandidat verfügbar (jede Seite also im Working Set eines Prozesses), so kann man davon ausgehen, dass ein Austausch schädlich ist (aufgrund der Überlastung des Systems). Ein Seitentausch würde nur eine aktive Seite treffen und damit weitere Seitenfehler nach sich ziehen. Besser ist es, in diesem Fall den anfordernden Prozess stillzulegen und erst dann wieder zu reaktivieren, wenn sich die Verhältnisse gebessert haben.

8.2.2 Einstellung der Fenstergröße h

Zur Beantwortung der Frage, welche Kriterien man heranziehen kann, um die Fenstergröße h optimal zu wählen, lassen sich drei Vorschläge machen:

Das Knie-Kriterium

Das **Knie-Kriterium** lautet folgendermaßen:

Wähle h so, dass $W(t,h) \cong m_{opt}$, wobei m_{opt} die Seitenzahl ist, welche zum primären Knie gehört. Das **primäre Knie** der Lifetime-Funktion ist der Kurvenwert einer Geraden (durch den Ursprung), die sich $L(m)$ von oben nähert und diese berührt. h heißt Abzisse des primären Knies genau dann, wenn

$$\frac{L(h)}{h} \geq \frac{L(h^*)}{h^*}$$

für alle h^* gilt (siehe Abbildung 8.6).

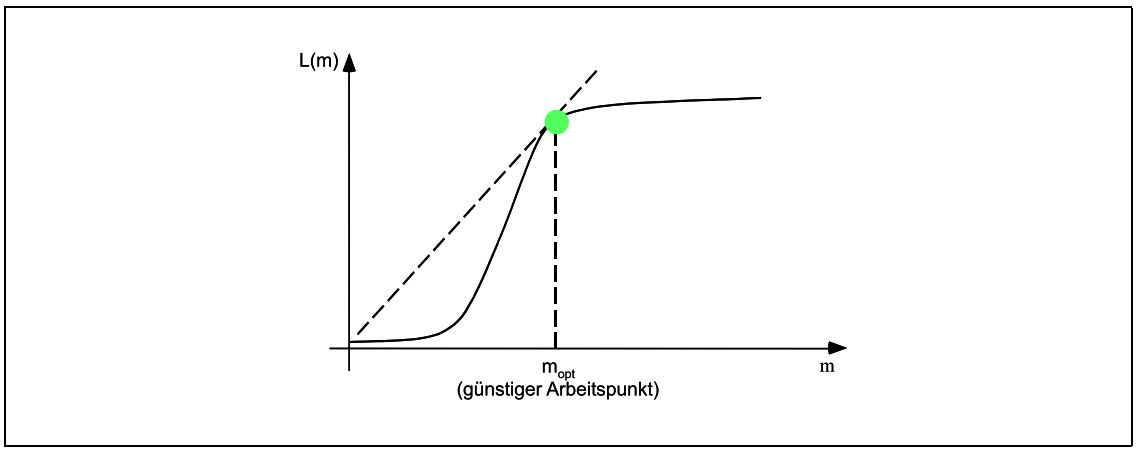


Abbildung 8.6: Knie-Kriterium

Das $L = S$-Kriterium

Für das **L=S-Kriterium** gilt:

Stelle h so ein, dass in etwa Lifetime = Swaptime gilt. Dieses Kriterium ergibt sich aus folgenden Überlegungen:

Der Gesamtdurchsatz D des Systems wird beschränkt durch:

a) Die mittlere Jobrate $1/T$, wobei T die mittlere Rechenzeit/Job ist.

b) Die Kapazität der Paging-Station, die die Seitenfehler bearbeitet.

Sei $a(m)$ die Seitenfehlerrate eines Jobs bei m zugeteilten Seiten und b die Bedienrate der Paging-Station (d.h. die Bediendauer eines Seitenfehlers beträgt $1/b$ Zeiteinheiten). Dann belegt jeder Job die Pagingstation für $T \cdot a(m)/b$ ZE und der Durchsatz wird durch $b/(T \cdot a(m))$ beschränkt (siehe Abbildung 8.7).

Somit ergibt sich

$$D \leq \frac{1}{T} \cdot \min\left(1, \frac{b}{a(m)}\right).$$

Mit

$$a(m) = \frac{1}{L(m)} = \frac{1}{\text{Lifetime}}$$

und

$$b = \frac{1}{S} = \frac{1}{\text{Swaptime}} \qquad \left(= \frac{1}{\text{Bediendauer}}\right)$$

ergibt sich schließlich

$$D \leq \frac{1}{T} \cdot \min\left(1, \frac{L(m)}{S}\right).$$

Dies ist gut erfüllt, wenn $L(m) \approx S$ gilt.

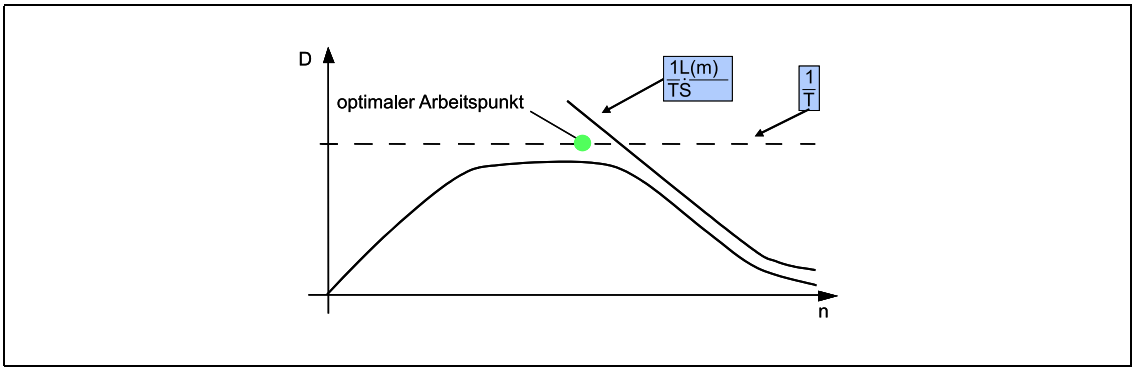


Abbildung 8.7: Das L=S-Kriterium

Eine leichte Verbesserung erreicht man noch, wenn $L(m) \approx c \cdot S$ gewählt wird, wobei c ein Erfahrungswert ist und etwa $1,3$ oder $1,5$ betragen sollte.

Das 50%-Kriterium

Das 50%-**Kriterium** besagt: Wähle h so, dass die Paging-Station zu ungefähr 50% ausgelastet ist. Man beachte, dass dieses Kriterium fast identisch zum L = S - Kriterium ist.

Nachweis:

Die Paging-Station kann als $M/M/1$-Warteschlangensystem angegeben werden. Das erste M bedeutet dabei, dass die Zeiten zwischen zwei Ankünften an der Paging-Station memoryless, d.h. negativ exponentialverteilt sind, das zweite M bedeutet Analoges für die Bedienzeitverteilung und die 1 steht für eine Bedienstation.

Für $M/M/1$ gilt nun:

$$\text{Auslastung} = \rho = \frac{\lambda}{\mu} < 1,$$

wobei λ der Parameter der Ankunftsverteilung und μ der Parameter der Bedienverteilung ist. Die mittlere Kundenzahl $\bar{N}$ berechnet sich bei $M/M/1$-Systemen wie folgt:

$$\bar{N} = \frac{\rho}{1-\rho}.$$

$\bar{N} = 1$ wird für $\rho = 1/2$ erreicht, also bei 50% Auslastung der Paging-Station. $\bar{N} = 1$ bedeutet jedoch auch, dass $L(m) = S$ ist. Realistisch gesehen, sind die Bedienzeiten der Paging-Station jedoch keineswegs memoryless, sondern nahezu konstant. Das Modell für diesen Fall lautet $M/D/1$, wobei D für deterministische Bedienung steht.

Für die mittlere Kundenzahl $\bar{N}$ in $M/D/1$-Systemen gilt:

$$\bar{N} = \frac{\rho}{1-\rho} - \frac{\rho^2}{2(1-\rho)}$$

also

$$\bar{N} = 1 \quad \Leftrightarrow \quad \rho = 2 \pm \sqrt{2} \approx 0,58.$$

Der Fall $\rho = 2 + \sqrt{2}$ kommt hierbei nicht in Frage, da die Auslastung ρ nie über 1 steigen kann. Die optimale Auslastung der Paging-Station, die sich mit der Verbesserung ergibt, liegt also bei $\approx 58\%$.

Ist die Größe des Working-Sets beschränkt, so stellt sich die Frage, nach welchen Kriterien frei werdende Seiten ersetzt werden. Eine einfache **Working-Set-Strategie** wäre, eine freie Seite beliebig auszuwählen. **WS-LRU** dagegen nutzt die bei fester Speicherzuteilung bewährte Strategie: Aus der Gesamtheit der frei werdenden Seiten wird diejenige verdrängt, die am längsten nicht genutzt wurde. Im Vergleich zu solchen **Fixed-Space-Strategien** zeigt sich, dass **Variable-Space-Strategien** wie **WS-LRU** i.A. deutlich bessere Ergebnisse bringen als selbst Fixed-OPT.

Ein Problem der Working-Set-Strategie liegt darin, dass sie aufwendig ist, da der aktuelle Working Set regelmäßig berechnet werden muss.

Vereinfachung:

Steuere zugeteilte Rahmen pro Prozess bzw. Multiprogramminggrad über Page-Fault-Frequency (PFF), d.h. die Seitenfehlerhäufigkeit des Programms:

1. Wenn PFF zu hoch ist, dann gib diesem Programm mehr Seiten. Ist dieses nicht möglich, da bereits alle Seiten belegt sind, so lagere das Programm aus.
2. Wenn PFF zu niedrig ist, dann gib Seite des entsprechenden Programms frei und starte evtl. ein zusätzliches Programm.

8.3 Die optimale Strategie VOPT

Gibt es nun (ähnlich wie im fixen Fall) eine optimale variable Strategie? Die Antwort lautet: Ja, und sie hat sogar einen Namen: **Variable-OPT** (VOPT). Zunächst sollte jedoch die Bedeutung von optimal in diesem Zusammenhang definiert werden:

Eine Variable-Space-Strategie heißt *optimal* genau dann, wenn die Strategie die Zahl der Seitenfehler bei gegebener mittlerer Zahl zugeteilter Seiten minimiert.

Betrachtet man nun einen Referenzstring zur Zeit t, so ergibt sich der Working Set $WS(t,h)$ (wie schon bekannt) aus dem Rückwärtsfenster:

$$WS(t,h) = RF(t,h) = \bigcup_{j=t-h+1}^{t} r_j$$

Für VOPT wird dagegen das Vorwärtsfenster benutzt:

$$VF(t,h) = \bigcup_{j=t+1}^{t+h} r_j$$

Ist zusätzlich M_t die Menge der Seitenrahmen, die einem Programm zur Zeit t zugeordnet sind, so lautet die Strategie VOPT wie folgt:

Sei r_t die zuletzt referenzierte Seite. r_t wird gehalten, falls r_t im Vorwärtsfenster $VF(t,h)$ enthalten ist. Ist r_t darin nicht enthalten, so wird die Seite sofort verdrängt, d.h. sie ist weder in $M_{t+1}, M_{t+2}, \ldots$ noch in M_{t+h} enthalten.

Zwei direkte Folgerungen daraus sind:

1. $VF(t,h) = RF(t+h,h)$
2. VOPT agiert bezüglich der Seitenfehler genauso wie WS. WS hält jedoch überflüssige Seiten noch h Zeiteinheiten länger als VOPT.

Der Nachweis, dass VOPT und WS genau dieselben Seitenfehler produzieren, ist einfach. Seien r_t und r_{t+u} $(u > 1)$ zwei aufeinanderfolgende Zugriffe auf dieselbe Seite. Dann gilt mit einer Fallunterscheidung:

a) $u > h$: VOPT wirft die Seite sofort raus (vor r_{t+1}). WS entfernt die Seite auch, jedoch erst zum Zeitpunkt r_{t+h} $(t+h < t+u)$, denn erst dann ist die Seite nicht mehr im Rückwärtsfenster.

b) $u \leq h$: Keine der Strategien lagert die Seite aus.

Im Fall a) müssen also beide Strategien die Seite erneut laden, während sie im Fall b) jeweils erhalten bleibt.

Trotzdem gibt es einen wichtigen Unterschied zwischen den Strategien, wie der schon eben angesprochene Fall a) vermuten lässt: Die mittlere zugeteilte Seitenzahl ist bei VOPT geringer als bei WS. Deutlich wird dies insbesondere bei **Phasenwechseln** von Programmen, d.h. in den Zeitbereichen, in denen Programme von einem lokalen Bereich in einen anderen wechseln. Während sich bei VOPT die Zahl der zugeteilten Seitenrahmen verringert, überschätzt WS die Anzahl wirklich benötigter Rahmen (siehe Abbildung 8.8).

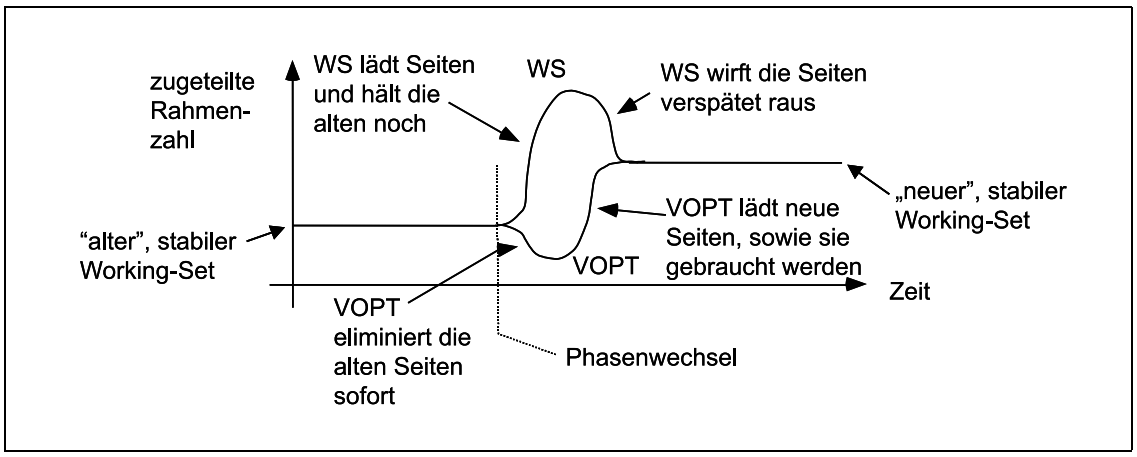


Abbildung 8.8: Verhalten von WS bzw. VOPT bei Phasenwechseln

Bisher bezogen sich die Kosten, mit der eine Strategie beurteilt wurde, ausschließlich auf die Zahl der Seitenfehler. Wie gerade gezeigt, ist aber auch die Einbeziehung von Kosten, die die Anzahl der gehaltenen Seiten beschreiben, sinnvoll. Es soll nun gelten:

$$\text{Kosten} = \text{Seitenfehlerkosten} + \text{Seitenhaltekosten}$$

Bei Fixed-Space-Strategien war diese Berücksichtigung übrigens nicht nötig, da in diesem Fall jedem Programm eine feste Anzahl von Seiten zugeteilt war, durch deren Freigabe kein weiteres Programm lauffähig wurde.

Es gilt nun folgender Satz:

SATZ	**Optimaler Paging Algorithmus**
	Bezüglich der neuen Kostenfunktion ist VOPT der optimale Paging-Algorithmus, falls die Fenstergröße $h = \frac{R}{U}$ gewählt wird. Dabei entspricht R den Kosten der Bearbeitung eines Seitenfehlers und U den Kosten für das Halten einer Seite pro Zeiteinheit.

Beweis:

Sei $\omega = r_1 r_2 \ldots r_t$ ein beliebiger Referenzstring, der M Seiten benötigt. Die Kosten C dieses Referenzstrings ergeben sich dann zu:

$$C = \underbrace{M \cdot R}_{\substack{\text{Kosten, die durch den} \\ \text{Erstzugriff entstehen}}} + \underbrace{\sum_{i=1}^{t} c_i}_{\substack{\text{Kosten, die nach Zugriff } r_i \text{ bis zum} \\ \text{nächsten Zugriff auf} \\ \text{diese Seite entstehen}}}$$

Die Kosten c_i berechnen sich dabei folgendermaßen:

Seien r_i und r_j zwei aufeinanderfolgende Zugriffe auf dieselbe Seite. Definiert man weiter $x_i := j - i$, so ergeben sich folgende zwei Fälle:

a) r_i wird zwischen i und j nicht aus dem Speicher geworfen $\Rightarrow c_i = x_i \cdot U$, d.h. die Seite wird x_i Zugriffe lang gehalten und es entstehen entsprechende Kosten (siehe Abbildung 8.9).

b) r_i wird zwischen i und j (im Schritt z_i, $0 \leq z_i < j$) aus dem Speicher geworfen $\Rightarrow c_i = z_i \cdot U + R$, d.h. die Seite wird z_i Zugriffe lang gehalten, danach entfernt und anschließend mit Kosten R neu geladen (für VOPT gilt immer $z_i = 0$) (siehe Abbildung 8.10).

Gegeben sei nun ein Fenster der Größe $h = R/U$ für VOPT. Für Fall a) muss offensichtlich gelten $h = R/U < x_i$.

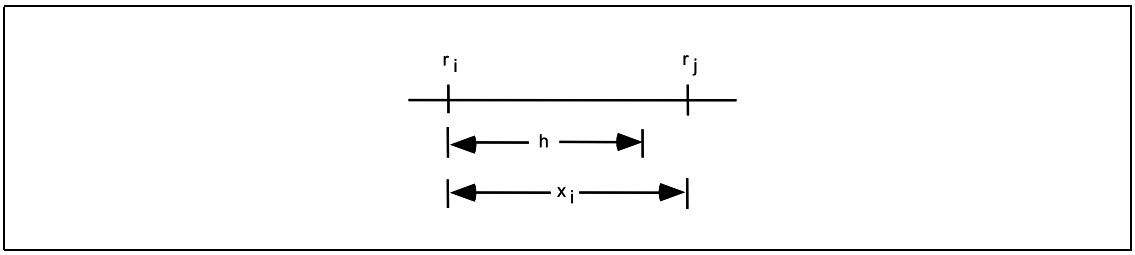


Abbildung 8.9: Optimaler Paging Algorithmus: Fall $h < x_i$

VOPT wirft r_i raus und lädt sie wieder nach $\Rightarrow c_i$ (VOPT) $= U \cdot 0 + R = R$. Vergleicht man VOPT mit einer beliebigen anderen Strategie A^*, welche die Seite hält, so ergibt sich $c_i(A^*) = U \cdot x_i > U \cdot h = R = c_i$ (VOPT), d.h. VOPT ist im Fall a) besser als jede andere Strategie A^*.

Für Fall b) gilt $h = R/U \geq x_i$.

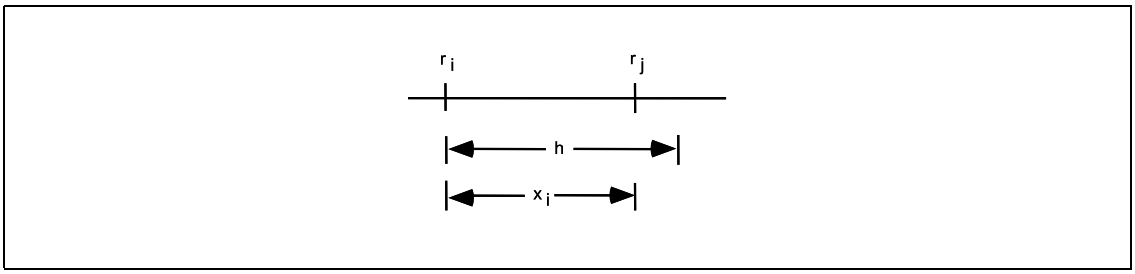


Abbildung 8.10: Optimaler Paging Algorithmus: Fall $h \geq x_i$

VOPT hält also $r_i \Rightarrow c_i(\text{VOPT}) = U \cdot x_i$. Vergleicht man VOPT wieder mit einer beliebigen anderen Strategie A^{**}, welche die Seite (im Schritt $z_i > i$) auslagert, so ergibt sich

$$c_i(A^{**}) = U \cdot z_i + R \quad > \quad R = U \cdot h \quad \geq \quad U \cdot x_i = c_i(VOPT)$$

d.h. VOPT ist auch im Fall b) besser als jede andere Stragegie A^{**}, q.e.d.

KAPITEL 9

Das Dateisystem

Da das Dateisystem für Otto Normaluser derjenige Teil des Betriebssystems ist, mit dem er am häufigsten Kontakt hat, darf ein Abriss über seine Funktionsweise hier nicht fehlen. Im folgenden Kapitel werden wir daher kurz, auf die wichtigsten Aspekte eingehen.

9.1 Das Datei-Konzept

Alle Rechneranwendungen sind darauf angewiesen, Informationen über einen längeren Zeitraum und unabhängig von einzelnen Prozessen speichern zu können. Hierzu stehen verschiedene Möglichkeiten zur Verfügung. Um davon abstrahieren zu können, führt man die **Datei** als logische Speichereinheit ein, die vom Betriebssystem auf einen nichtflüchtigen Speicher, z.B. eine **Festplatte**, abgebildet wird.

DEFINITION

Datei

Eine **Datei** ist eine mit Namen versehene Sammlung zusammengehöriger Informationen, welche auf einem Hintergrundspeicher liegt.

Die Art der in einer Datei gespeicherten Informationen reicht dabei von Quell-, Objekt- und ausführbaren Dateien über numerische Daten, Texte und Bilder bis hin zu einkommenssteuererklärungsrelevanten Daten, wobei natürlich eine Datei je nach ihrem Typ unterschiedliche Strukturen aufweisen kann.

Einer Datei sind stets verschiedene **Attribute** zugeordnet. Sicherlich am wichtigsten ist der Dateiname, der dem Nutzer ein Bezugnehmen auf die Datei erlaubt. Oft wird an den Namen noch eine **Typinformation** (**Extension**) angehängt, anhand derer das Betriebssystem erkennen kann, um welche Art von Datei es sich handelt. Typische Dateinamen lauten damit etwa `prog.c` oder `filesys.doc`. Weitere Attribute können beispielsweise die Größe einer Datei, ihre Position im Speicher, Informationen über Zugriffsrechte oder -zeit, Datum und Benutzeridentitäten sein. Derartige Attribute werden allesamt im Rahmen einer **Verzeichnisstruktur** auf der Festplatte gespeichert.

Zur vollständigen Charakterisierung einer Datei als abstrakten Datentyp sind noch die **Operationen auf Dateien** anzugeben. Unverzichtbar sind hier das *Erzeugen* und *Löschen* von Dateien, *Lese-* und *Schreiboperationen* sowie die *Umpositionierung* von Zeigern für

Lese- oder Schreibzugriffe innerhalb einer Datei. Die Möglichkeit, neue Informationen an das Dateiende *anzuhängen*, Dateien *umzubenennen*, zu *kopieren* usw. werden durch von Betriebssystem zu Betriebssystem variierende Mechanismen realisiert.

Die meisten derartigen Operationen machen es erforderlich, dass das Betriebssystem zunächst im entsprechenden Verzeichnis nach dem Dateinamen sucht. Diese Suche wird dann überflüssig, wenn der Benutzer gezwungen wird, vor dem Zugriff auf eine Datei diese zu öffnen und sie analog nach dem Ende der Benutzung wieder zu schließen. Das System kann dann eine Liste geöffneter Dateien (**Open-File-Table**) führen und damit eine aufgerufene Datei wesentlich schneller auffinden.

Der Zugriff selbst kann dann auf unterschiedliche Weise geregelt sein. Am einfachsten ist der so genannte **sequenzielle Zugriff**, bei dem eine Eintragung nach der anderen abgearbeitet wird. Der **direkte Zugriff** ermöglicht es dagegen, in der Datei an beliebigen Stellen zu lesen bzw. zu schreiben. Dies ist günstiger, wenn man unmittelbaren Zugang zu großen Mengen von Informationen wünscht, also etwa in Datenbanken. Viele Betriebssysteme unterstützen beide Zugriffsweisen.

9.2 Verzeichnisstruktur

Da das Dateisystem eines Rechners einen beträchtlichen Umfang erreichen kann, wird man versuchen, es irgendwie sinnvoll zu organisieren. Gewöhnlich geht man dazu in zwei Schritten vor: Zunächst unterteilt man das Dateisystem in verschiedene **Partitionen** (siehe Abbildung 9.1). Teilt man eine Festplatte in mehrere Partitionen auf, dann kann man die so entstandenen Bereiche als getrennte Speicher behandeln. Partitionen können sich aber auch über mehrere Festplatten erstrecken und dienen dann zur logischen Gruppierung der Festplatten.

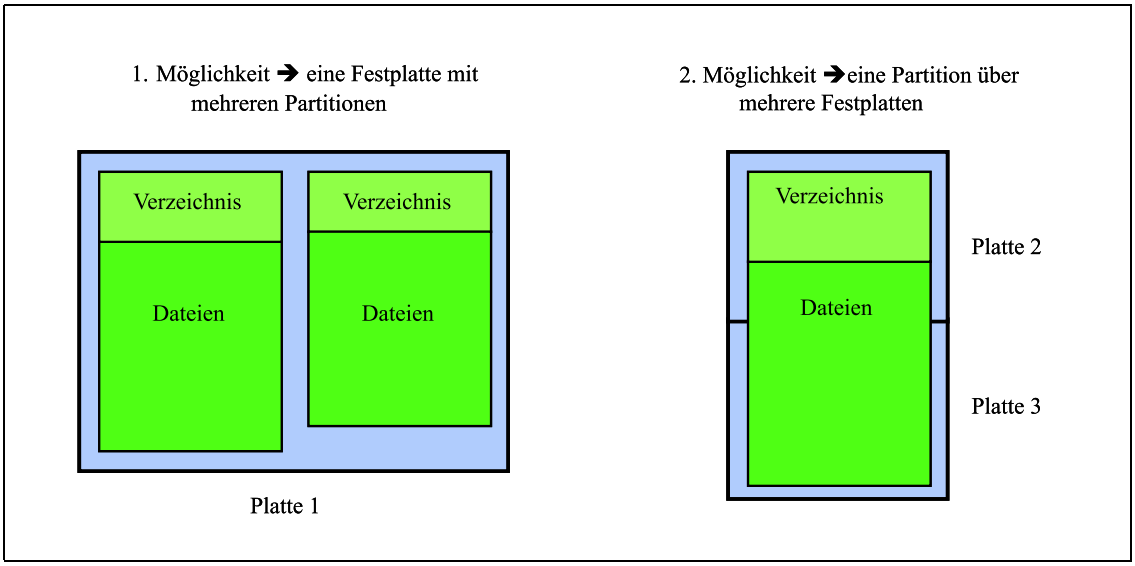


Abbildung 9.1: Partionen und Verzeichnisse

Jede Partition enthält dann ihrerseits Informationen über die in ihr enthaltenen Dateien. Diese Informationen stehen in **Verzeichnissen** (**Directories**) und umfassen die Attribute aller Dateien, die in der betreffenden Partition gespeichert sind und insbesondere natürlich deren Namen. Das Verzeichnis selbst kann auf vielerlei Weise strukturiert werden, wobei die auf einem Verzeichnis möglichen **Operationen** zu berücksichtigen sind, beispielsweise

das *Suchen* nach einer Datei, ihr *Erzeugen*, *Löschen* und *Umbenennen* sowie das *Auflisten* des Verzeichnisses. Nützlich ist auch, Zugriffsmöglichkeiten auf das gesamte Dateisystem vorzusehen.

9.2.1 Single-Level-Verzeichnis

Die einfachste Verzeichnisstruktur ist das Single-Level-Verzeichnis, d.h. alle Dateien sind in ein und demselben Verzeichnis enthalten (siehe Abbildung 9.2).

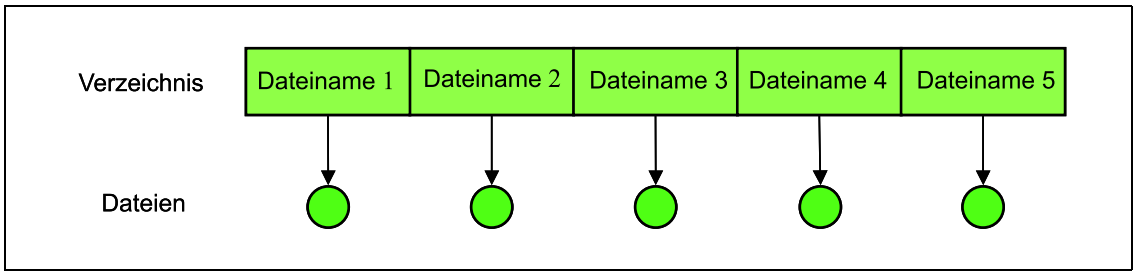


Abbildung 9.2: Single-Level-Verzeichnis

Sobald aber eine größere Menge an Dateien vorliegt oder mehrere Benutzer beteiligt sind, werden die beschränkten Möglichkeiten dieses Ansatzes schnell deutlich. Beispielsweise müssen alle Dateinamen eindeutig sein. Dies kann nicht nur im Falle mehrerer Benutzer problematisch werden, sondern auch aufgrund der Tatsache, dass viele Betriebssysteme die Länge der verwendbaren Dateinamen beschränken.

9.2.2 Two-Level-Verzeichnis

Um die Verwirrung über die Dateinamen, wie sie unter mehreren Benutzern leicht auftritt, in den Griff zu bekommen, weist man am einfachsten jedem Benutzer ein eigenes Verzeichnis zu und gelangt so zu einer Verzeichnisstruktur, die zwei Ebenen umfasst (siehe Abbildung 9.3).

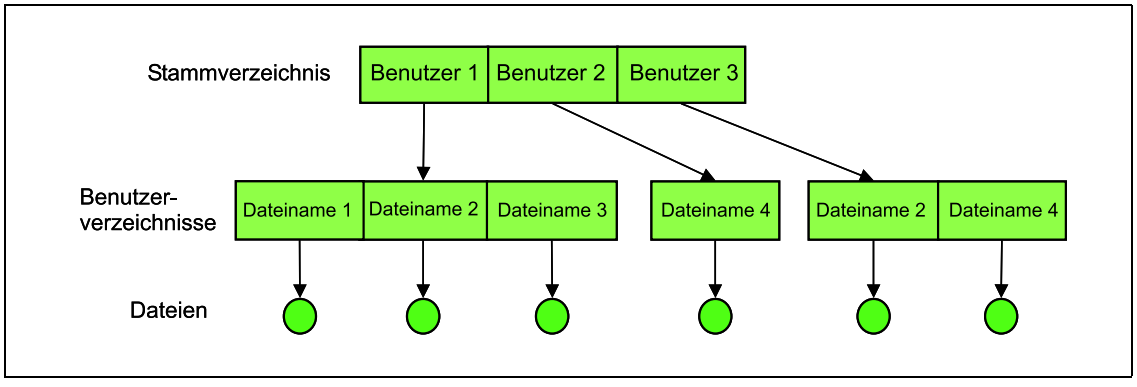


Abbildung 9.3: Two-Level-Verzeichnis

Greift ein Benutzer auf eine seiner Dateien zu, so wird nur sein eigenes Verzeichnis danach durchsucht. Daher können jetzt zwei Benutzer gleich benannte Dateien speichern. Andererseits kann ein Benutzer aber auch auf Dateien eines anderen zugreifen, indem er statt des einfachen Dateinamens einen **Pfad** angibt, der aus Benutzer- und Dateiname besteht (z.B. */Benutzer3/Dateiname2*).

9.2.3 Verzeichnisbäume

Diese Idee lässt sich natürlich verallgemeinern und führt zu einer baumartigen Verzeichnisstruktur. Jeder Benutzer kann damit sein Verzeichnis in beliebige Unterverzeichnisse aufteilen und entsprechend seine Dateien strukturiert ablegen. Der Verzeichnisbaum hat ein **Stammverzeichnis** (Root-Directory), und jeder Dateiname entspricht dann einem eindeutigen Pfad von der Wurzel durch alle Unterverzeichnisse hin zur betreffenden Datei (siehe Abbildung 9.4).

Bei der tagtäglichen Arbeit am Rechner befindet sich ein Benutzer stets in einem aktuellen Verzeichnis, das sinnvollerweise möglichst viele der Dateien enthält, die für ihn gerade von Interesse sind. Gibt er nämlich einen einfachen Dateinamen an, so wird nur das aktuelle Verzeichnis daraufhin durchsucht. Um eine Datei außerhalb des aktuellen Verzeichnisses anzusprechen, muss ihr gesamter Pfad angegeben oder das aktuelle Verzeichnis gewechselt werden. Pfade kann man dabei auf zwei Weisen angeben: der **absolute Pfad** beginnt bei der Wurzel und geht abwärts bis zur betreffenden Datei, während der **relative Pfad** vom aktuellen Verzeichnis ausgeht.

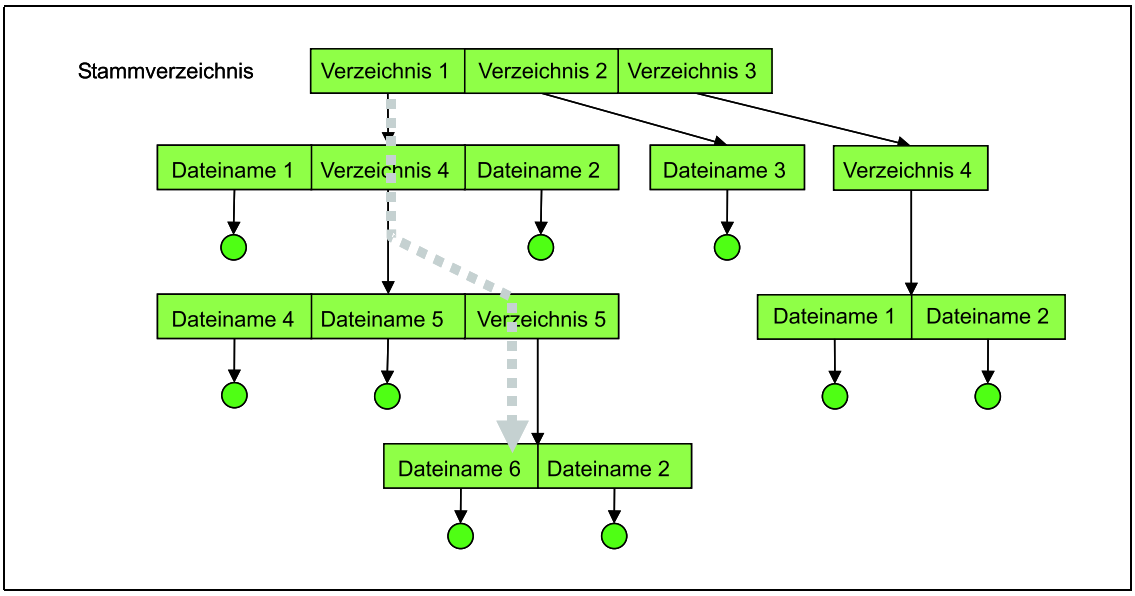


Abbildung 9.4: Verzeichnisbaum

Interessant ist die Frage, wie das Löschen eines Verzeichnisses vonstatten gehen soll. Befinden sich keine Dateien oder Unterverzeichnisse darin, so kann man einfach den entsprechenden Eintrag im darüberliegenden Verzeichnis löschen. Ist das Verzeichnis aber nicht leer, so gibt es zwei Möglichkeiten:

- Bei manchen Betriebssystemen kann man ein Verzeichnis erst dann entfernen, wenn es ganz leer ist, was leicht zu einem beträchtlichen Aufwand führen kann.
- Gefährlicher, aber effizienter, ist es, beim Löschen eines Verzeichnisses automatisch alle darin enthaltenen Dateien und Unterverzeichnisse mit zu entfernen.

9.2.4 Verzeichnisse mit azyklischem und allgemeinem Graphen

Wenngleich die eben besprochene Baumstruktur einem Benutzer leicht den Zugriff auf Dateien eines anderen Benutzers ermöglicht, können in speziellen Situationen andere Struk-

turen noch effektiver sein. Wenn zum Beispiel zwei Leute an einem gemeinsamen Projekt sitzen und die dafür relevanten Dateien zusammen in einem Unterverzeichnis abgelegt sind, so ist es vorstellbar, dass jeder der beiden dieses Unterverzeichnis in seinem eigenen Verzeichnisbaum haben möchte. Ein solches gemeinsam genutztes Verzeichnis erscheint also im Dateisystem an zwei (oder mehr) Stellen, existiert aber in Wirklichkeit nur einmal. Die beiden Programmierer arbeiten also nicht etwa an zwei unabhängigen Exemplaren derselben Dateien, sondern wenn einer Änderungen ausführt, so sind diese sofort auch dem anderen ersichtlich.

Dies ist allerdings bei Baumstrukturen nicht möglich, wohl aber, wenn das Dateisystem als (gerichteter) **azyklischer Graph** aufgebaut ist (siehe Abbildung 9.5).

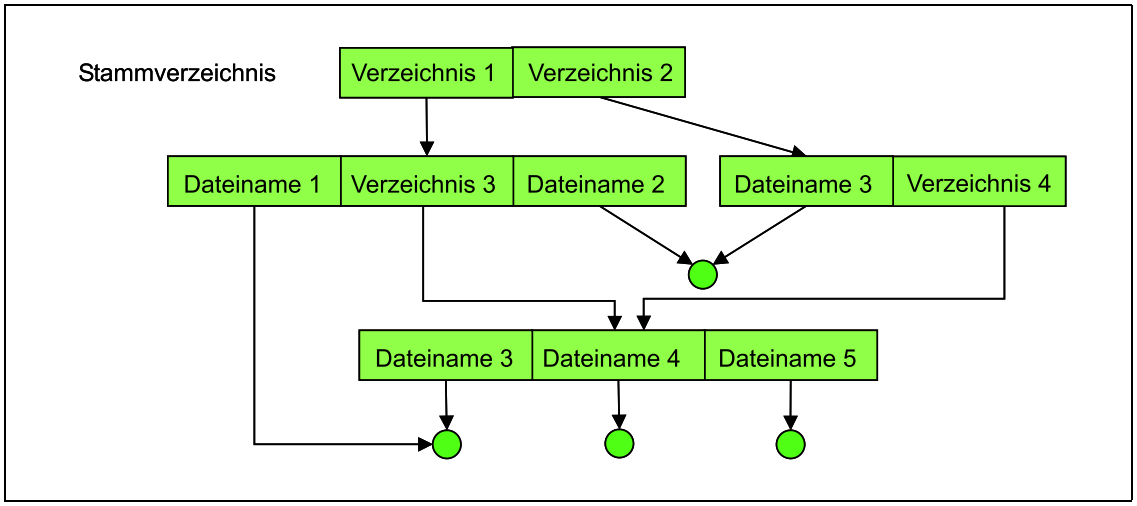


Abbildung 9.5: Verzeichnisstruktur als azyklischer Graph

Eine solche Struktur ist flexibler als ein Baum, aber auch komplexer. Insbesondere gibt es keine eindeutigen absoluten Pfade mehr, auch das Löschen eines Verzeichnisses wird komplizierter. Ein schwerwiegendes Problem ist aber auch die Frage, wie man sicherstellen will, dass tatsächlich keine Zyklen vorkommen. Nur dann nämlich bleiben die Algorithmen zur Durchsuchung des Gesamtverzeichnisses nach einer Datei halbwegs einfach. Sobald Zyklen auftreten, kann es vorkommen, dass ein schlecht gebauter Suchalgorithmus in eine Endlosschleife läuft; auch das Löschen eines Verzeichnisses wird in einem allgemeinen Graphen noch ein Stückchen komplizierter.

9.3 Implementierung von Dateisystemen

Nach den bisher eher theoretischen Betrachtungen wenden wir uns nun langsam Fragen der Implementierung von Dateisystemen zu. Wir wissen bereits, dass sich das Dateisystem auf einem Hintergrundspeicher befindet, der große Mengen von Daten auf Dauer abspeichern kann. Festplatten bieten sich hierfür an, da sie zum einen modifizierte Daten sofort an der Stelle speichern können, an der die ursprünglichen standen, und da sie außerdem den direkten Zugriff auf beliebige Stellen einer Datei ermöglichen. Um den Datentransport zwischen Hauptspeicher und Platte effizient zu gestalten, werden Daten blockweise übertragen. Jeder **Block** besteht dabei aus einem oder mehreren **Sektoren**.

Das Design eines Dateisystems stellt einen vor zwei unterschiedliche Fragen: Wie soll das Dateisystem für den Benutzer aussehen, und welche Algorithmen und Strukturen sind notwendig, um dieses logische Dateisystem auf den physikalischen Speicher abzubilden?

Normalerweise ist ein **Dateisystem aus mehreren Schichten** aufgebaut, wobei jede Schicht die von den unterhalb liegenden Schichten angebotenen Dienste dazu nutzt, ihrerseits höheren Schichten Dienste zur Verfügung zu stellen (siehe Abbildung 9.6).

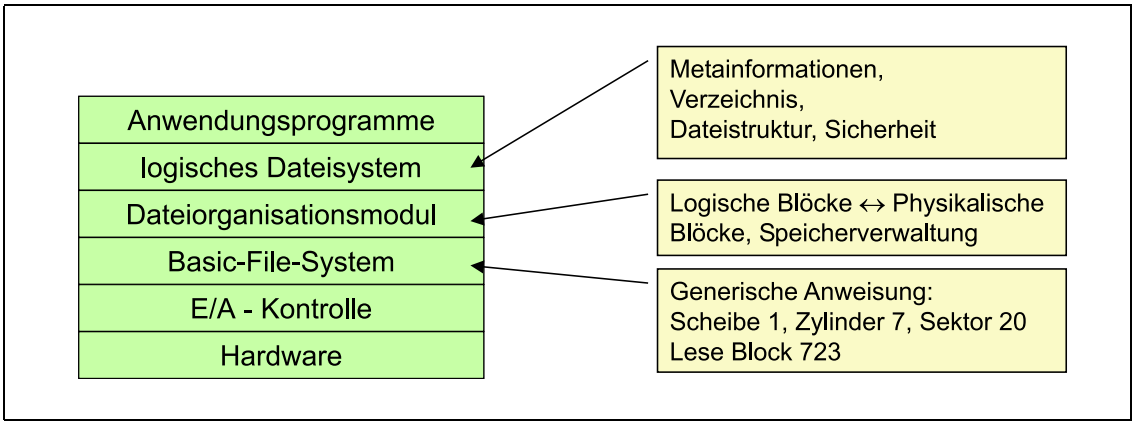


Abbildung 9.6: Schichten eines Dateisystems

Die unterste Schicht eines Dateisystems, die **E/A-Kontrolle**, besteht im Wesentlichen aus **Treibern**, die die technische Seite der Datenübertragung zwischen Hauptspeicher und Plattensystem regeln. Das **Basic-File-System** veranlasst die Treiber, physikalische Blöcke auf die Festplatte zu schreiben bzw. von dort zu lesen. In der nächsthöheren Ebene findet sich das **Dateiorganisationsmodul**, das die Verbindung zwischen den logischen Dateiblöcken und den physikalischen Blöcken herstellt und insbesondere dazu in der Lage ist, je nach verwendeter Speicherbelegungsstrategie logische Blockadressen in physikalische zu übersetzen. Das logische Dateisystem schließlich verwendet die oben erläuterte Verzeichnisstruktur dazu, das Dateiorganisationsmodul mit den nötigen Informationen zu einem gegebenen (symbolischen) Dateinamen zu versorgen. Um beispielsweise eine neue Datei zu erzeugen, ruft ein Anwendungsprogramm das **logische Dateisystem** auf (siehe Abbildung 9.7). Dieses liest das entsprechende Verzeichnis in den Speicher, erledigt die Neueintragung und schreibt die Modifikation auf die Festplatte zurück. Auch der E/A-Zugriff auf eine Datei erfolgt über das logische Dateisystem. Damit dieses aber nicht bei jedem Zugriff das gesamte Verzeichnis nach der fraglichen Datei durchsuchen muss, verwendet man meist die bereits erwähnte Open-File-Tabelle, die alle notwendigen Informationen über die derzeit offenen Dateien enthält.

Dass eine Datei geöffnet werden muss, bevor man auf sie zugreifen kann, wissen wir schon. Analog dazu ist ein (zusätzliches) Dateisystem zu **mounten**, bevor es den Prozessen zur Verfügung steht. Hierzu muss das Betriebssystem die Stelle innerhalb der bestehenden Dateistruktur erfahren, an der das neue Dateisystem einzuhängen ist (siehe Abbildung 9.8).

9.4 Belegungsstrategien

Die Möglichkeit, auf Festplatten direkt und sequenziell zugreifen zu können, gibt uns ein gewisses Maß an Flexibilität für die Implementierung der Dateispeicherung. Die Grundfrage lautet, in welcher Weise der Speicherplatz, den die Platte zur Verfügung stellt, auf die Dateien aufgeteilt wird, um einerseits den Speicher gut auszunutzen und andererseits einen schnellen Zugriff auf die Dateien sicherzustellen. Weitverbreitet sind vor allem drei Strategien: die zusammenhängende, die verkettete und die indizierte Belegung.

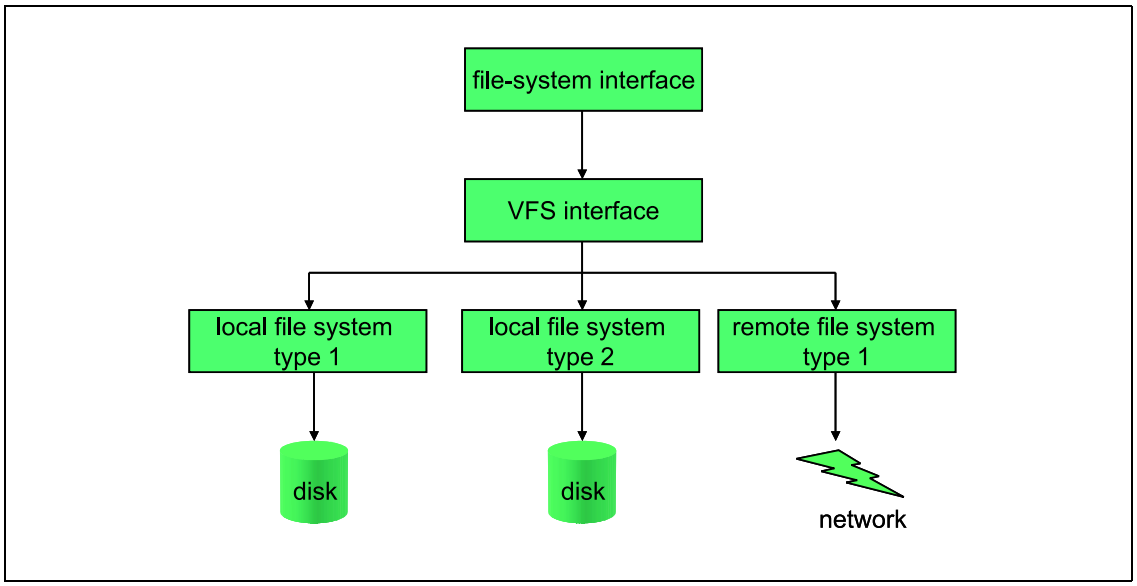


Abbildung 9.7: Virtuelles/Logisches Dateisystem

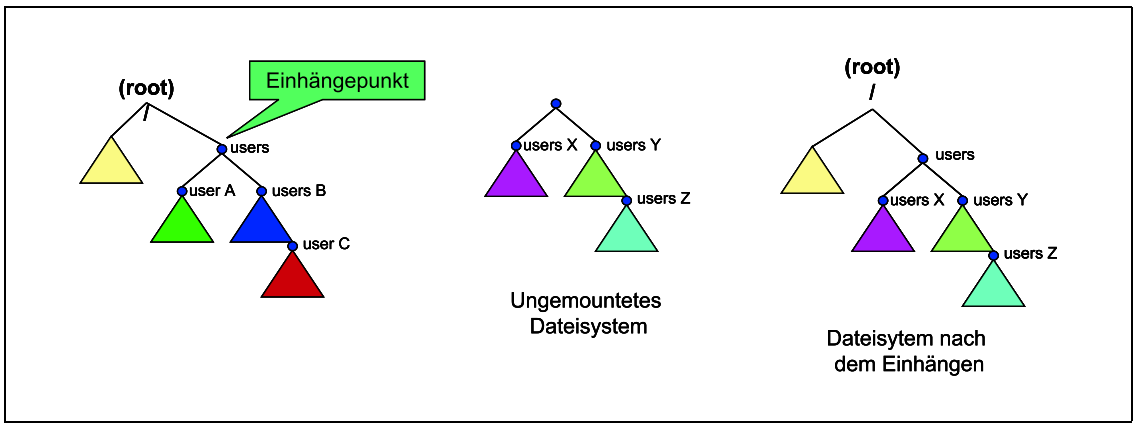
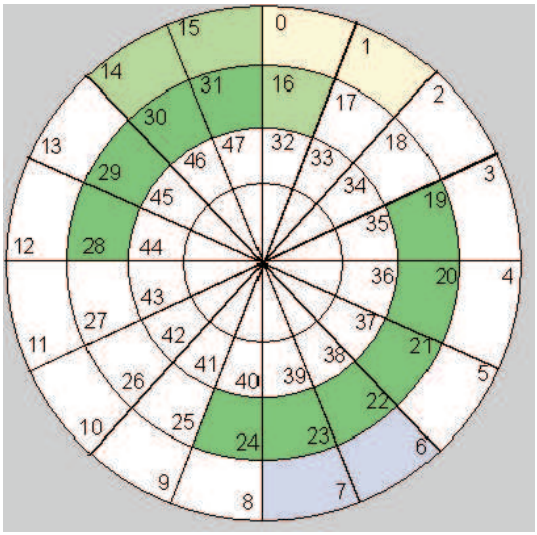


Abbildung 9.8: Einhängen von Dateisystemen

9.4.1 Zusammenhängende Belegung

Bei dieser Methode belegt jede Datei eine Reihe zusammenhängender Blöcke auf der Festplatte (siehe Abbildung 9.9). Die Belegung ist festgelegt, sobald die Adresse des ersten Blocks auf der Platte sowie die Länge der Datei (in Blöcken) bekannt ist. Der Zugriff auf eine zusammenhängend abgelegte Datei ist sowohl sequenziell als auch direkt auf einfache Weise möglich. Beim sequenziellen Zugriff genügt es, sich an die Adresse des zuletzt referenzierten Blockes zu erinnern. Will man direkt auf den (logischen) Block i der Datei zugreifen und beginnt diese beim Block b der Festplatte, so greift man einfach auf den (physikalischen) Block $b + i$ zu.

Ein Hauptproblem bei diesem Vorgehen ist die Frage, wie man genügend Speicherplatz für eine neue Datei ausfindig macht. Derartige Fragen haben wir bereits früher im Abschnitt über dynamische Speicherbelegung diskutiert. Damals stellten wir fest, dass **First-Fit** und **Best-Fit** die am meisten verbreiteten Strategien für dieses Problem sind. Natürlich laufen diese Algorithmen auf **externe Fragmentierung** des Speicherplatzes hinaus: Durch Ablegen und Entfernen der Dateien wird im Laufe der Zeit der freie Speicherplatz in kleine Stücke zerbrochen, sodass irgendwann u.U. selbst das größte freie Stück für eine neu zu



Verzeichnis

Datei	Start	Länge
Name 1	0	2
Name 2	14	3
Name 3	19	6
Name 4	28	4
Name 5	6	2

Ähnlich wie „Segmentierung“ mit allen Vor- und Nachteilen

Abbildung 9.9: Zusammenhängende Belegung eines Datenträgers

speichernde Datei nicht mehr ausreicht.

Ein weiteres ernsthaftes Problem ist die Frage, wie viel Platz eine Datei denn überhaupt braucht. Denn um ihr genügend Platz zuweisen zu können, muss die Größe einer Datei zum Zeitpunkt ihrer Erzeugung bekannt sein. Reserviert man ihr zu wenig Platz, dann ist es u.U. nicht möglich, die Datei später zu erweitern. Dies führt dann entweder zum Abbruch des entsprechenden Prozesses mit allen Konsequenzen, insbesondere wird der Benutzer beim Neustart auf Nummer sicher gehen und den Speicherbedarf deutlich überschätzen, was im Endeffekt auf eine mehr oder weniger heftige Speicherplatzverschwendung hinausläuft, oder dazu, dass für die erweiterte Datei ein neues, größeres Loch gesucht und die bisherigen Daten dorthin kopiert werden müssen. In diesem Fall arbeitet das System trotz des aufgetretenen Problems wenigstens weiter, jedoch unter einem gewissen Geschwindigkeitsverlust.

Selbst wenn der totale Speicherbedarf für eine Datei im Voraus bekannt ist, kann die Vorwegbelegung ineffizient sein. Hat man nämlich eine Datei, die langsam über einen größeren Zeitraum, z.B. ein Monat oder ein Jahr, stetig anwächst, so ist es nicht unbedingt sinnvoll, den Endbedarf von Anfang an zu reservieren, denn dann bleiben große Mengen an Speicherplatz über lange Zeit ungenutzt, ein Effekt, den man als **interne Fragmentierung** bezeichnet.

9.4.2 Verkettete Belegung

Die Methode der verketteten Belegung behebt die aufgetretenen Probleme der zusammenhängenden Belegung. Eine Datei ist hier als verkettete Liste von Blöcken realisiert, wobei diese Blöcke willkürlich über die ganze Platte verstreut sein können. Das Verzeichnis enthält dann einen Zeiger auf den ersten und einen auf den letzten Block der Datei, und jeder Block enthält einen Zeiger auf den nächsten Block der Datei (siehe Abbildung 9.10).

Die Erzeugung einer neuen Datei besteht einfach in einer neuen Eintragung ins Verzeichnis, wobei die Zeiger zu `nil` (was einer leeren Datei entspricht) und die Dateilänge zu 0 initialisiert werden. Eine Schreiboperation kann dann bewirken, dass ein freier Plattenblock

gefunden werden muss und mittels eines Zeigers ans Ende der bisherigen Datei angehängt wird. Beim Lesen der Datei folgt man einfach den Zeigern von Block zu Block.

Bei der verketteten Belegung kommt es weder zu externer Fragmentierung noch zu Verschwendung von Speicherplatz. Jeder freie Block auf der Platte kann genutzt werden, und es ist auch nicht nötig, die Länge einer Datei am Anfang verbindlich anzugeben. Vielmehr kann eine Datei weiter und weiter wachsen, solange freie Blöcke auf der Platte vorhanden sind.

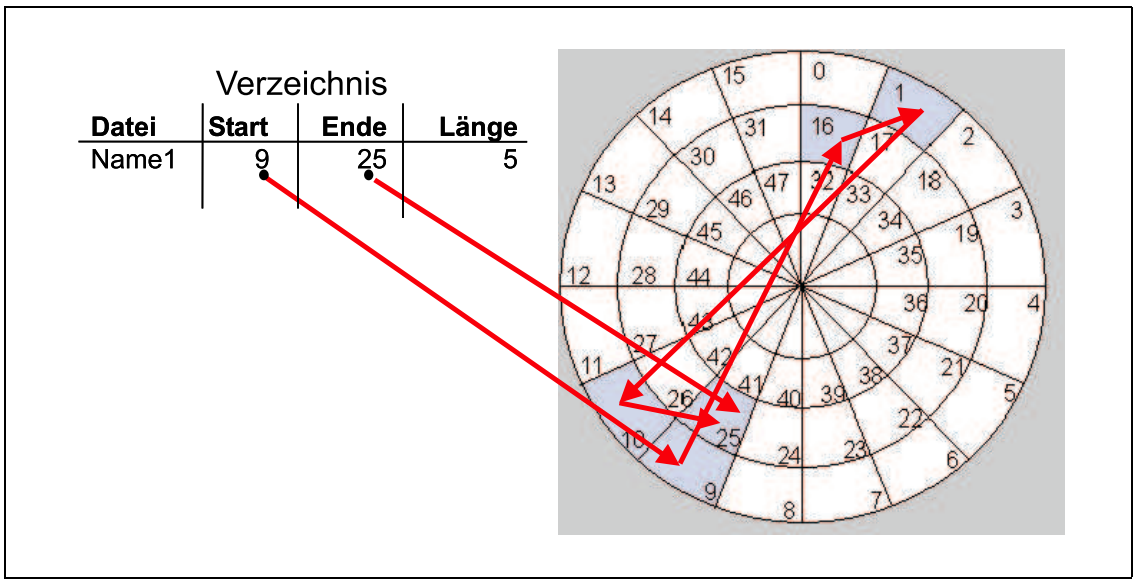


Abbildung 9.10: Verkettete Belegung auf dem Datenträger

Allerdings hat diese Strategie auch Nachteile. Vor allem ist sie nur für sequenzielle Zugriffe geeignet. Will man nämlich den i-ten Block einer Datei finden, so ist man gezwungen, am Beginn der Datei zu starten und den Zeigern solange zu folgen, bis man am fraglichen Block angekommen ist. Ein weiterer Nachteil ist, dass die Zeiger selbst Speicherplatz verbrauchen, den man sonst anderweitig nutzen könnte. Dieses Problem löst man üblicherweise dadurch, dass man mehrere Blöcke zu einem Cluster zusammenfasst und diesen als Speichereinheit anstelle der Blöcke verwendet. Man braucht weniger Platz für die Zeiger, die Abbildung der logischen auf die physikalischen Blöcke bleibt einfach, und dennoch erhöht sich der Systemdurchsatz. Freilich geht dies auf Kosten einer höheren internen Fragmentierung, da Platz verschwendet wird, sobald ein Cluster nur teilweise belegt ist.

Ein weiterer Nachteil dieses Verfahrens wird offensichtlich, wenn man die Frage nach der Zuverlässigkeit stellt. Da eine Datei – durch Zeiger zusammengehalten – über die ganze Platte verstreut gespeichert ist, hat der Verlust oder die Beschädigung eines einzigen Zeigers schnell fatale Auswirkungen, ebenso wie die versehentliche Verwendung eines falschen Zeigers aufgrund etwa eines Software-Fehlers. Sich dagegen abzusichern bringt oft großen Overhead mit sich.

Eine wichtige Variante beruht auf der Verwendung einer Dateibelegungstabelle **File-Allocation-Table** (FAT) (siehe Abbildung 9.11). Hierbei befindet sich am Anfang jeder Partition eine Tabelle, in der für jeden Block der Partition ein Feld vorgesehen ist. Diese Tabelle funktioniert ähnlich wie eine verkettete Liste: Ein Dateieintrag im Directory enthält die Nummer m des ersten (physikalischen) Blocks, mit dem die gespeicherte Datei beginnt. An m-ter Stelle der Tabelle ist dann die Nummer des folgenden Blocks angegeben. Diese Kette geht weiter bis zum letzten Block der Datei, der in der Tabelle einen

speziellen `EOF`-Wert zugeordnet bekommt. Freie Blöcke werden in der Tabelle durch den Wert 0 gekennzeichnet.

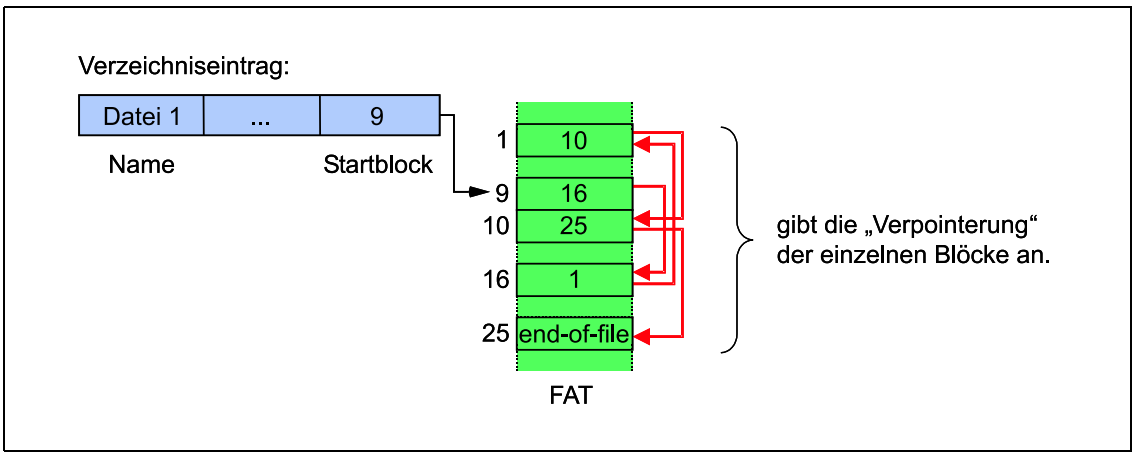


Abbildung 9.11: File-Allocation-Table

Zu bedenken ist allerdings, dass die Verwendung einer **FAT** für die Platte erhöhten Suchaufwand mit sich bringt, da der Lese-/Schreibkopf ständig zwischen der Tabelle und den entsprechenden Blöcken hin- und herspringen muss. Andererseits wird der direkte Zugriff dadurch wesentlich vereinfacht, da die Plazierung eines willkürlich herausgegriffenen Blocks schnell anhand der FAT festgestellt werden kann. Eine weitere Schwierigkeit besteht darin, dass ein Verlust der FAT zum Verlust der gesamten Datei führt.

9.4.3 Indizierte Belegung

Wir haben gesehen, dass die verkettete Belegung die Schwierigkeiten der zusammenhängenden Belegung in den Griff bekommt, aber nicht für einen effizienten direkten Zugriff geeignet ist. Die indizierte Belegung löst dieses Problem dadurch, dass sie alle Zeiger an einem Platz, dem **Indexblock**, zusammenfasst. Jede Datei hat ihren eigenen Indexblock, der aus lauter Blockadressen auf der Platte besteht. Dabei zeigt der i-te Eintrag im Indexblock auf den i-ten (physikalischen) Block der gespeicherten Datei. Das Directory enthält dann nur noch die Adresse des Indexblocks. Um den i-ten Dateiblock zu lesen, schaut man an der i-ten Stelle des Indexblocks nach und kann dann direkt auf die gewünschte Stelle in der Datei zugreifen (siehe Abbildung 9.12).

Somit unterstützt indizierte Belegung einen direkten Zugriff, ohne unter externer Fragmentierung zu leiden, da jeder freie Block auf der Platte zur Speicherung verwendet werden kann. Was verschwendet wird, ist Speicherplatz für den Indexblock, insbesondere wenn ein kompletter Block reserviert werden muss, obwohl die zugehörige Datei relativ klein ist.

Wie groß sollte also ein Indexblock sein? Da jede Datei einen Indexblock besitzt, erscheint es ratsam, diese nicht zu groß zu machen. Zu kleine Indexblöcke wiederum können u.U. nicht alle für eine Datei benötigten Zeiger aufnehmen, sodass für die Speicherung großer Dateien Extramechanismen vorzusehen sind: Beispielsweise könnte man eine Verkettung von Indexblöcken vorsehen. Üblicherweise würde dann einer Datei ein Indexblock zugeordnet, an dessen letzter Stelle entweder ein `nil`-Zeiger (kleine Datei) oder ein Zeiger auf einen weiteren Indexblock (große Datei) zu finden ist. Oder man führt eine neue Ebene von Indexblöcken ein, deren Einträge auf Indexblöcke zeigen, die ihrerseits dann auf Dateiblöcke zeigen (siehe Abbildung 9.13). Dieses Schema ist hierarchisch beliebig weit ausbaufähig. Eine dritte Möglichkeit kombiniert die beiden vorherigen Ansätze, indem sie

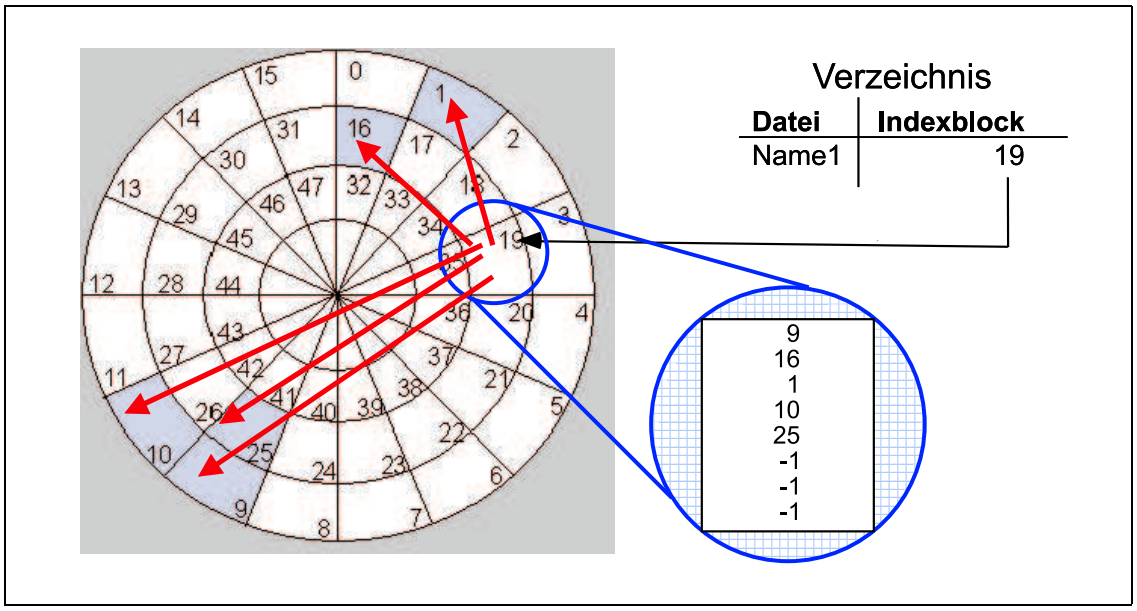


Abbildung 9.12: Indizierte Belegung auf dem Datenträger

einen Indexblock aufteilt in einen Bereich, der Adressen von Dateiblöcken enthält, und einen kleineren Bereich, in dem die Adressen weiterer Indexblöcke stehen können.

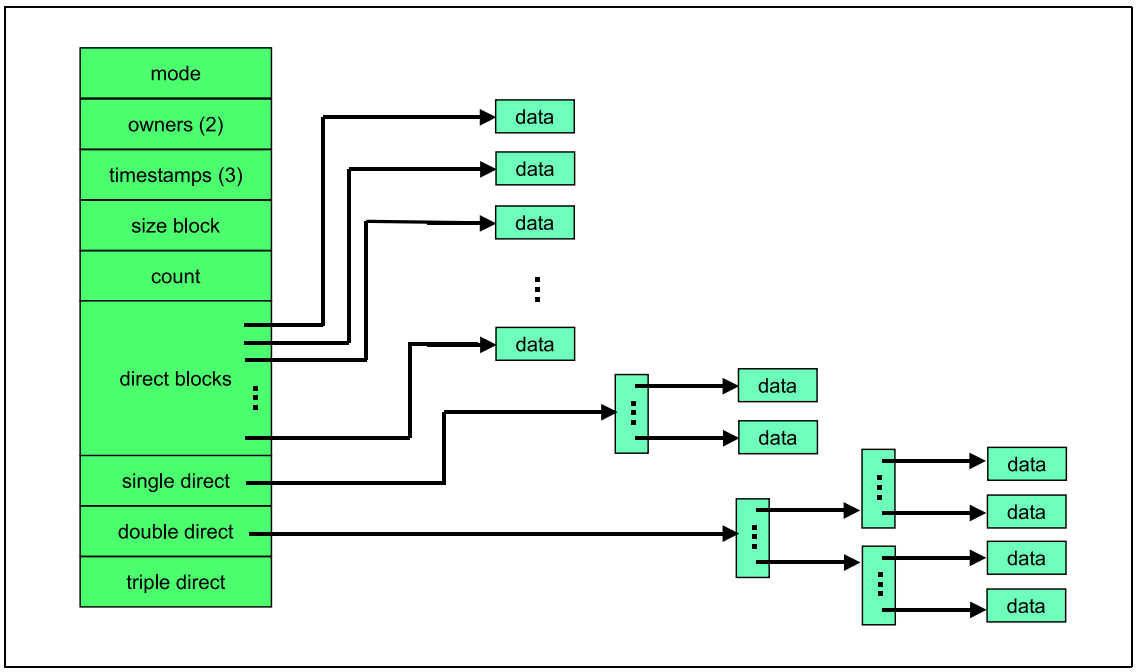


Abbildung 9.13: Kombination aus indizierter Belegung und kaskadierter Indizierung

9.5 Speicherplatzverwaltung

Da auf einer Platte Speicherplatz nur in begrenztem Ausmass vorhanden ist, sieht man sich dazu gezwungen, diesen – so es geht – wiederzuverwenden. Um über den freien Speicherplatz auf dem Laufenden zu sein, bedient man sich einer Tabelle, der so genannten

Free-Space-List, in der alle Plattenblöcke verzeichnet stehen, die nicht durch eine Datei oder ein Verzeichnis belegt sind. Bei der Erzeugung einer Datei sucht man dann in dieser Tabelle nach entsprechend viel Speicherplatz, weist diesen der neuen Datei zu und wirft ihn zugleich aus der Free-Space-List heraus. Umgekehrt wird anlässlich eines Dateilöschens der von ihr bisher belegte Platz in die Tabelle aufgenommen.

Trotz ihres Namens wird die Free-Space-List nicht immer als Liste implementiert, sondern zum Beispiel als *Bit-Vektor*. Hierbei wird jeder Block durch ein Bit repräsentiert. Ist der Block frei, so wird das Bit auf 1 gesetzt, andernfalls auf 0. Das ist relativ einfach, zudem ist es auf effiziente Weise möglich, den ersten freien Block oder n freie Blöcke hintereinander ausfindig zu machen. Man kann aber auch alle freien Blöcke auf der Platte zu einer *verketteten Liste* zusammenfügen und einen Zeiger auf ihren ersten Block an einer besonderen Stelle auf der Platte speichern. Dieser erste Block enthält dann einen Zeiger auf den nächsten freien Block usw. Eine Modifikation davon besteht in der *Gruppierung* freier Blöcke. Hierbei sind im ersten freien Block n Adressen gespeichert. Die ersten $n-1$ dieser Adressen zeigen auf tatsächlich freie Blöcke, während die letzte Adresse auf einen Block zeigt, der seinerseits $n-1$ Adressen freier Blöcke und eine weitere Blockadresse enthält usw. Diese Implementierung ist deshalb so wichtig, da mit ihrer Hilfe auf einen Blick die Adressen einer großen Anzahl freier Blöcke herausgefunden werden können. Schließlich kann man auch noch ausnutzen, dass (insbesondere bei der zusammenhängenden Belegung oder beim Clustern) in der Regel eine größere Anzahl zusammenhängender Blöcke gleichzeitig frei wird, sodass es genügt, die Adresse des ersten derartigen Blocks parat zu haben und dann durch einfaches *Zählen* die Anzahl n freier zusammenhängender Blöcke festzustellen, anstatt etwa eine Liste von n einzelnen Blöcken zu verwalten.

9.6 Implementierung von Verzeichnissen

Da die Auswahl von Algorithmen zur Verzeichnisverwaltung großen Einfluss auf Effizienz, Leistungsfähigkeit und Zuverlässigkeit des Dateisystems haben kann, soll abschließend noch kurz hierauf eingegangen werden.

9.6.1 Lineare Liste

Am einfachsten implementiert man ein Verzeichnis als **lineare Liste**. Um einen bestimmten Eintrag zu finden, muss man diese Liste dann der Länge nach durchsuchen, was sich ziemlich zeitaufwendig gestalten kann. Bei der Erzeugung einer neuen Datei ist zunächst die Liste daraufhin zu durchsuchen, ob der vorgesehene Dateiname noch nicht verwendet worden ist. Dann hängt man den Zeiger auf die neue Datei einfach am Ende der Liste an. Beim Löschen einer Datei wird ihr Eintrag aus der Liste entfernt. Den so gewonnenen freien Platz kann man auf dreierlei Weise nutzen: Entweder man markiert ihn als frei oder man trägt ihn in eine Liste freier Verzeichnisplätze ein. Die dritte Alternative ist es, den letzten Eintrag des Verzeichnisses an die freigewordene Stelle zu kopieren und dadurch die Liste zu verkürzen.

Der wirkliche Nachteil dabei ist die Linearität der Suche nach einer bestimmten Datei, weil Informationen aus dem Verzeichnis sehr oft gebraucht werden und eine langsame Implementierung an dieser Stelle sich schmerzlich bemerkbar machen kann. Viele Betriebssysteme implementieren daher einen Software-Cache, um die zuletzt aufgerufenen Einträge aus dem Verzeichnis zu speichern. Man kann die durchschnittliche Suchzeit auch durch Verwendung einer sortierten Liste verringern, aber der Implementierungsaufwand und die

Komplexität von Such- und Verwaltungsalgorithmen steigt dadurch natürlich beträchtlich.

9.6.2 Hash-Tabelle

Ein anderer Ansatz zur Implementierung von Verzeichnissen verwendet eine Hash-Tabelle. Unter einem **Hash** versteht man dabei eine Abbildungsfunktion:

$$h: \quad \{\text{Schlüssel}\} \rightarrow \{\text{Speicherposition in der Hash-Tabelle}\}$$

Die Abbildung h sollte leicht zu berechnen sein und den verfügbaren Speicherraum möglichst gleichmäßig ausnutzen. Injektivität wird nicht verlangt, d.h. unter Umständen ergibt die Anwendung von h auf zwei verschiedene Schlüssel dasselbe Resultat. In unserem Fall wandelt h jeden Dateinamen (= Schlüssel) in eine Position innerhalb einer **Hash-Tabelle** um (siehe Abbildung 9.14). Der Tabelleneintrag an der ermittelten Position ist dann ein Zeiger auf die Stelle innerhalb des immer noch als lineare Liste vorliegenden Verzeichnisses, an der die fragliche Datei (samt ihren Attributen) zu finden ist.

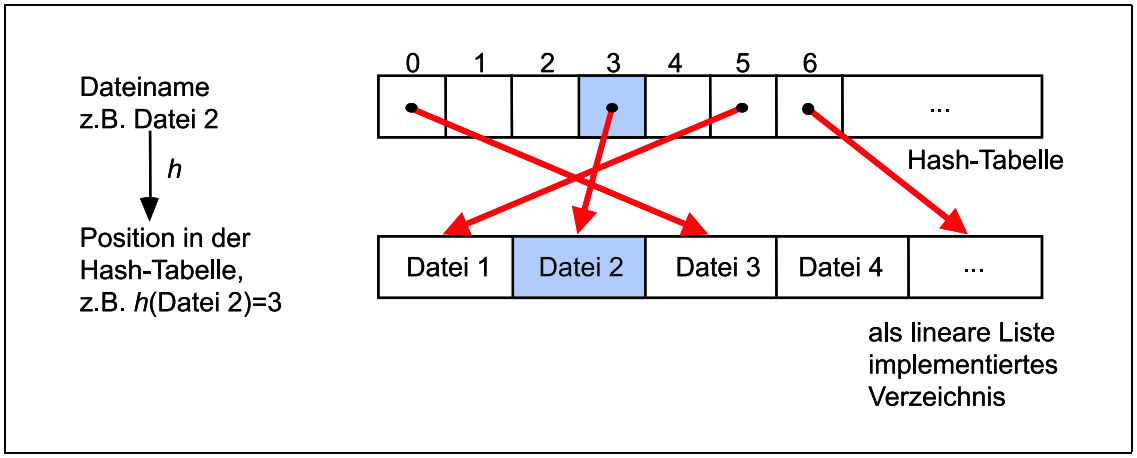


Abbildung 9.14: Beispiel einer Hash-Tabelle

Dadurch wird die Suchzeit bei geschicktem Vorgehen enorm reduziert. Auch das Einfügen und Entfernen von Einträgen ist relativ problemlos, wenngleich Vorkehrungen für den Fall einer Kollision getroffen werden müssen. Nachteilig macht sich bei diesem Ansatz bemerkbar, dass die Hash-Tabelle nur einen begrenzten Umfang besitzt und die Hash-Funktion obendrein von dieser Tabellengröße abhängt, d.h. wenn sich die bislang verwendete Tabelle als zu klein erweist und man auf eine größere umsteigt, so ist auch die Hash-Funktion komplett neu zu berechnen.

TEIL IV

Ergänzendes

KAPITEL 10

Binden und Laden von Programmen

Programme werden i.A. nicht an einem Stück programmiert, sondern bestehen aus einzelnen **Modulen**, die untereinander Beziehungen aufweisen. Um aus diesen Modulen ein lauffähiges Programm zu erzeugen, müssen sie in einer geeigneten Weise miteinander verknüpft werden. Diese Aufgabe übernimmt gewöhnlich ein Programm namens **Binder** oder **Linker**. Die Aufgabe des Binders ist es, die Module in der gewünschten Reihenfolge zusammenzufassen. Die zunächst symbolischen Bezüge zwischen den Bezeichnern müssen dabei konkretisiert bzw. aufgelöst werden. Hat der Binder seine Arbeit beendet, muss das Programm zur Ausführung in den Hauptspeicher gebracht werden. Diese Aufgabe wird vom **Lader** übernommen.

Zunächst stellt sich die Frage, in welcher Programmphase gebunden werden soll. Denkbar wären folgende Möglichkeiten:

Programmphase	**Binden günstig?**
Währen der Programmierung	Nein, da die Module dann nicht mehr unabhängig sind.
Wenn Quellcode komplett	Nein, da gleiche Module bei Wiederverwendung mehrfach übersetzt werden müssten.
Übersetzung	Nein, aus demselben Grund.
Wenn Module übersetzt vorliegen (Linkage Editor)	Günstig, da die Möglichkeiten zur Zusammensetzung flexibel bleiben und jetzt genügend Zeit vorhanden ist.
Beim Laden (Linkage Loader)	Gebräuchlich, jedoch mit dem Nachteil, dass man vor dem Start eines Programms das Binden abwarten muss.
Ausführung	Nein, obwohl sehr flexibel würde die Geschwindigkeit der Programme zu sehr leiden.

Anhand eines vereinfachten Beispiels wollen wir die Arbeitsweise eines Linkage-Editors studieren. Dabei lehnen wir uns an den Binder eines IBM-Großrechners an. Die durchzuführenden Schritte sind aber prinzipiell in allen Bindern nahezu identisch. Als Eingabe erhält der Linkage-Editor die zu bindenden **Objektmodule**. Dies sind bereits einzeln übersetzte Unterprogramme sowie eventuelle Steuerkommandos. Die Arbeitsweise

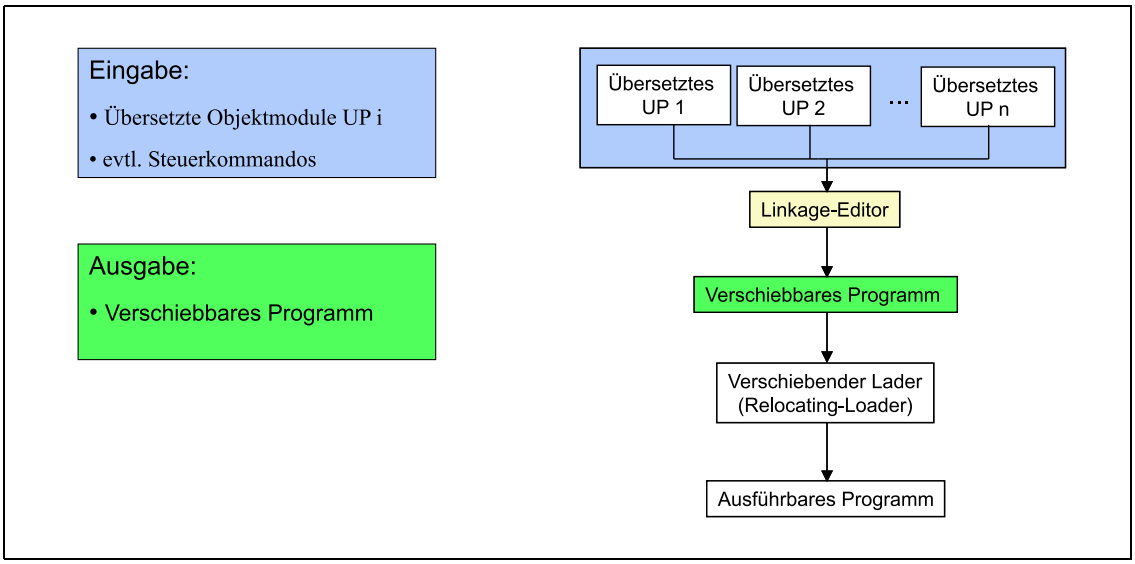


Abbildung 10.1: Prinzip des Linkage-Editors

des **Linkage-Editors** wird in Abbildung 10.1 veranschaulicht. Die Ausgabe des Linkage-Editors besteht dann aus einem lauffähigen Programm, also einem zusammengebundenen Objekt, welches durch einen **Relocating-Loader** geladen und ausführbar gemacht wird. Der generelle Aufbau eines Objekts (Objektmodul) ist in Abbildung 10.2 dargestellt. Das **External-Symbol-Dictionary** (ESD) ist eine Liste der im Teil `Text` vorkommenden externen Symbolen. Im **Relocation-Dictionary** (RLD) stehen diejenigen Bereiche im Text, die besonders behandelt werden müssen.

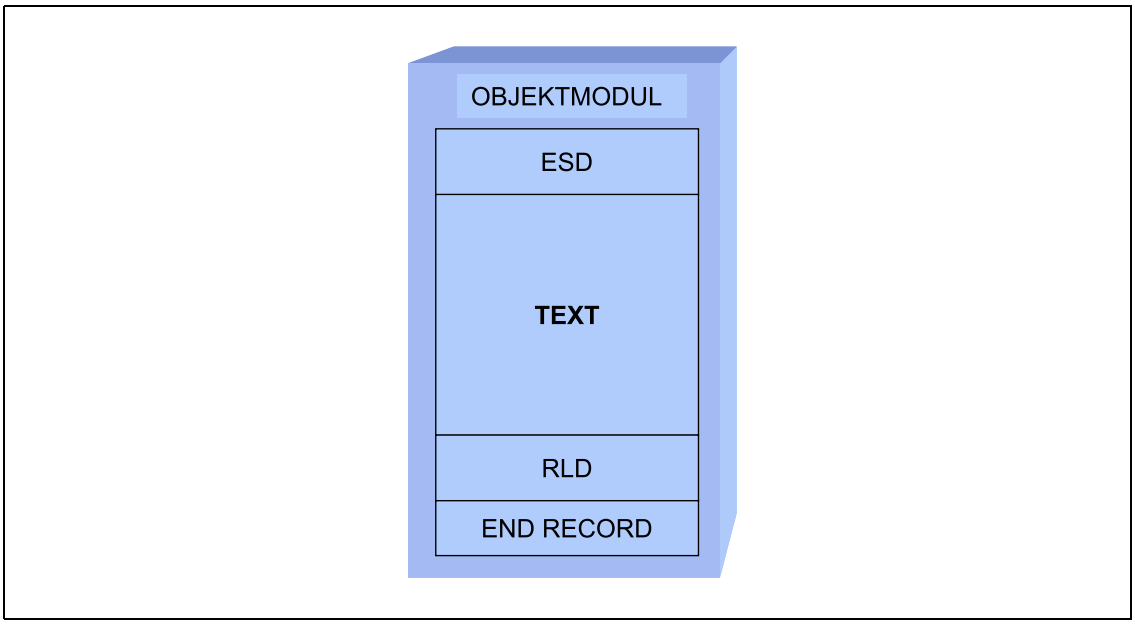


Abbildung 10.2: Prinzipieller Aufbau eines Objektmoduls

Das External-Symbol-Dictionary

Ein **externes Symbol** ist entweder ein externer Name oder eine externe Referenz. Unter einem externen Namen versteht man den Namen, unter dem das Modul von außen

erreichbar ist. Dies kann zum einen der Name des Unterprogramms sein (Name of Control Section) oder ein Label, welches eine symbolisch markierte Einsprungstelle darstellt (Entry Point). Eine externe Referenz ist ein Name, der einen Aufruf nach außen bewirkt, also einen Bezug zu allen externen Namen eines anderen Moduls herstellt (siehe Abbildung 10.3).

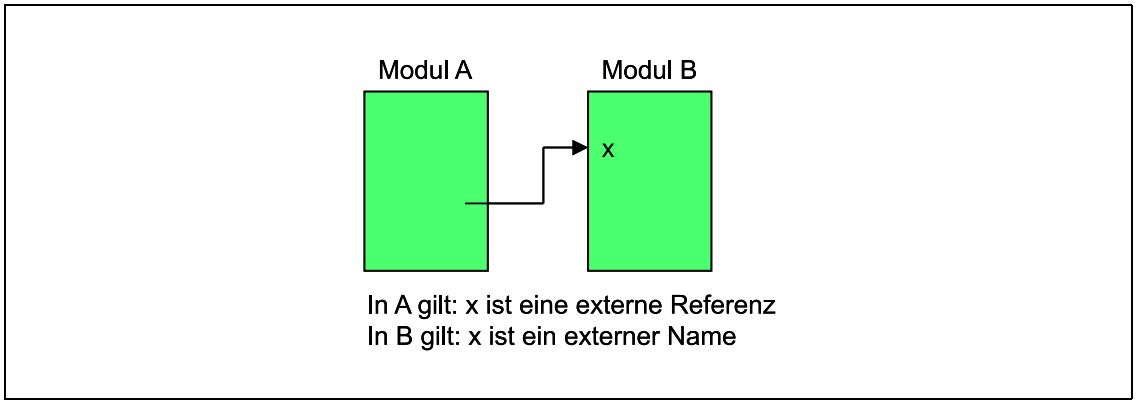


Abbildung 10.3: Beispiel einer externen Referenz

Die Aufgabe des Binders ist es, die eben beschriebenen Bezüge konkret herzustellen. Wenden wir uns nun dem Aufbau der ESD-Einträge und des Relocation-Dictionary zu. Der ESD-Eintrag sieht bei unserem Beispielbinder wie folgt aus:

Name	Typ	XXXXX	XXXXX

Die beiden letzten Felder sind dabei vom Typ des Eintrags abhängig. Die drei wichtigsten Typen sollen kurz vorgestellt werden:

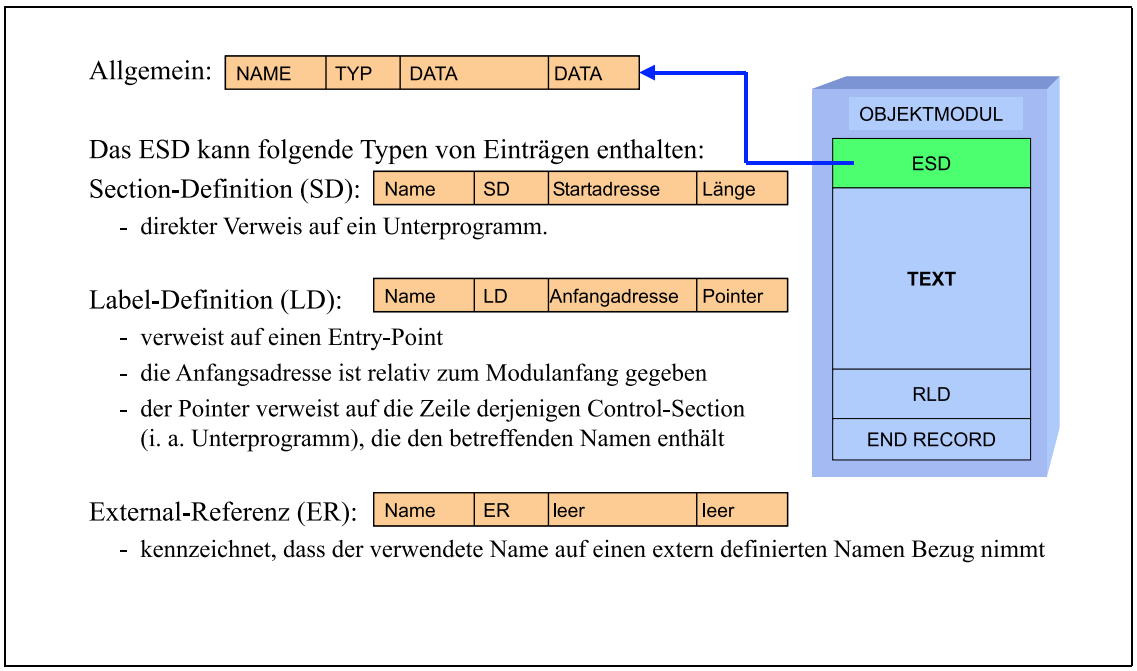


Abbildung 10.4: ESD Definition

Section Definition (SD): ist ein direkter Verweis auf ein Unterprogramm.

Label Definition (LD): Ein ESD-Eintrag vom Typ Label Definition verweist auf einen Entry Point. Die Anfangsadresse ist relativ zum Modulanfang. Der Pointer verweist auf die Zeile des ESD-Eintrags derjenigen Control Section (i.A. Unterprogramm), welche den betreffenden Namen enthält.

External Reference (ER): Mit diesem Eintrag wird lediglich gekennzeichnet, dass das verwendete Symbol auf ein extern definiertes Bezug nimmt.

Abbildung 10.4 stellt die einzelnen Möglichkeiten nochmal im Überblick dar.

Das Relocation-Dictionary

Das **Relocation-Dictionary** (RLD) enthält alle Adressen, bei denen beim Binden bzw. Laden noch Anpassungen vorgenommen werden müssen. Dies ist beispielsweise bei Adressbefehlen der Fall. Ein RLD-Eintrag hat den folgenden Aufbau:

Relocation Pointer	Position Pointer	Flag	Address

Der `Relocation Pointer` zeigt auf den entsprechenden ESD-Eintrag. Der `Position Pointer` verweist auf den ESD-Eintrag der Control Section, in der die betreffende Konstante definiert ist. Das `Flag` zeigt die Art eines Befehls an, und in `Address` steht der relative Abstand der Konstanten zum Programmbeginn. Befehle, die eine Verschiebung erfordern, sind beispielsweise:

Befehl	**Bedeutung**	**Wirkung**
DC A(X)	Define Constant Address	Inhalt dieser Zelle ist zur Ausführungszeit die aktuelle Adresse von X, wobei X ein lokaler Name ist.
DC V(X)	Define Constant Variable	Inhalt ist die aktuelle Adresse von X, jedoch ist X hier ein externes Symbol.

Die verschiedenen Einträge sollen nun an einem Beispiel erläutert werden. Dazu betrachten wir die zwei Module ALPHA und BETA aus Abbildung 10.5.

Der Binder (Linkage-Editor) erhält nun als Eingabe die Textmodule und jeweils die zugehörigen ESD und RLD. Daraus produziert er ein gemeinsames Modul mit Adressen, die relativ zueinander richtig positioniert sind. Ferner wird eine Composite-ESD aus allen Modulen erstellt (siehe Abbildung 10.6). Das gebundene Modul ist in Abbildung 10.7 dargestellt.

Bis jetzt wurde ein Modul erzeugt, das lediglich relative Adressen enthält. Dieses Modul muss noch an eine freie Stelle im Hauptspeicher geladen werden. Bei diesem Ladevorgang sind die relativen Adressen in die endgültigen absoluten Adressen umzurechnen. Diese Aufgaben übernimmt der Lader. Man unterscheidet zwischen absoluten und verschiebenden Ladern. Der absolute Lader transportiert das Programm unverändert an eine fest vorgegebene Adresse im Hauptspeicher. Der verschiebende Lader (Relocating-Loader) bestimmt zunächst einen ausreichend großen freien Speicher und platziert das Programm dann an

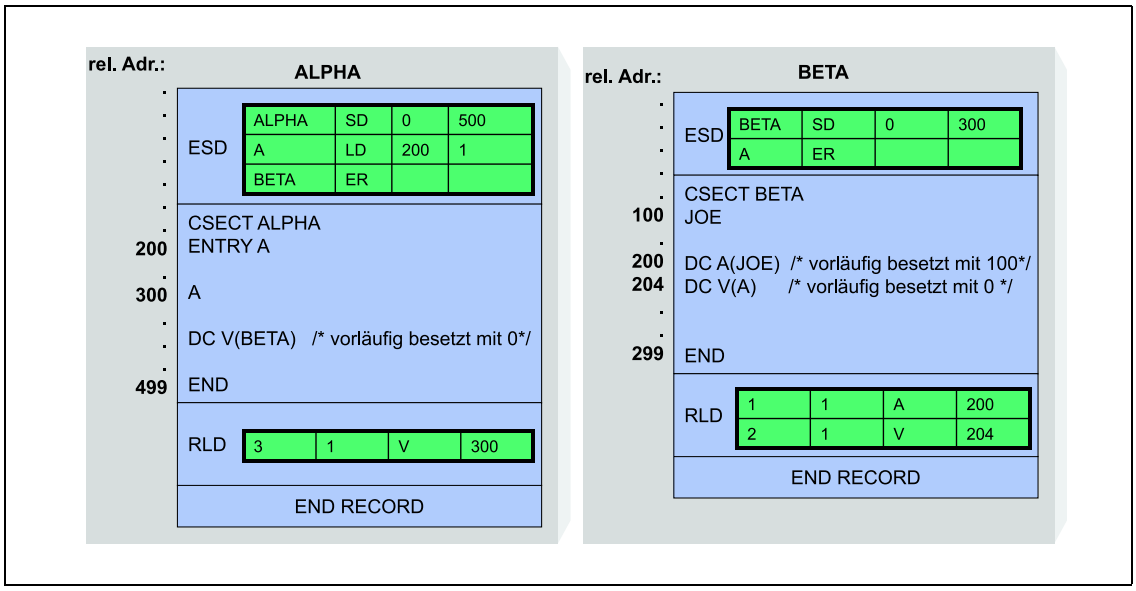


Abbildung 10.5: Beispiel Module Alpha und Beta

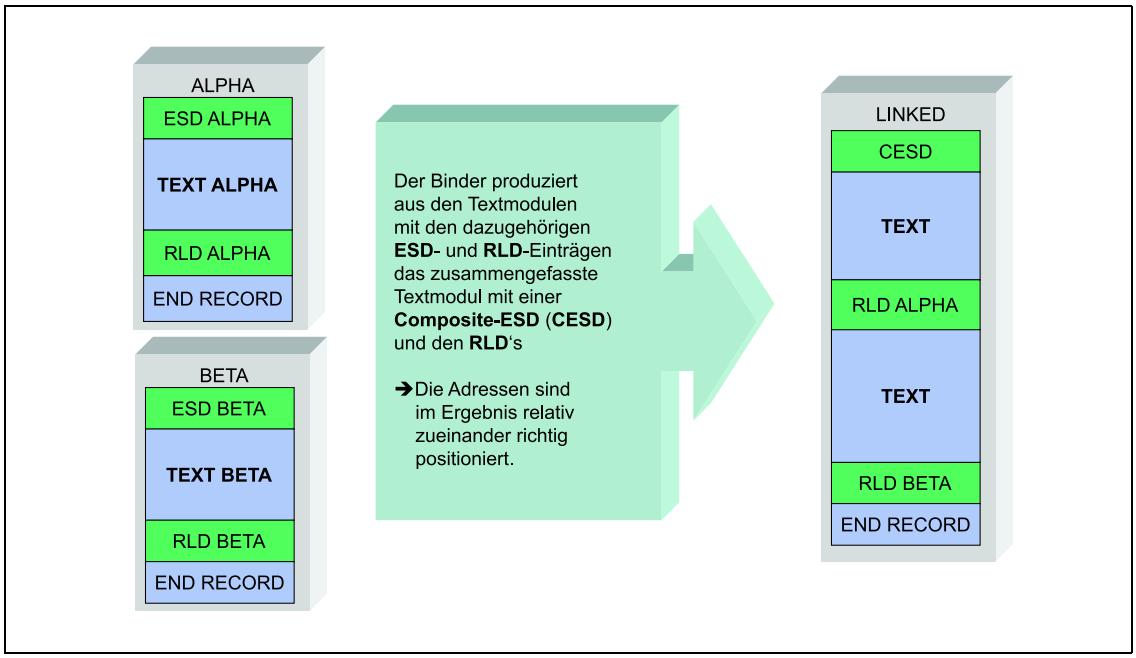


Abbildung 10.6: Prinzipieller Aufbau eines Composite ESD

diese Stelle. Die Adressen im Programm müssen mit Hilfe der Basisadresse zuzüglich der relativen Adresse berechnet werden. Wir wenden jetzt den Vorgang des verschiebenden Ladens auf unser gebundenes Beispielmodul an. Dabei gehen CESD und RLD verloren. Im Beispiel ist ein zusammenhängender Speicherbereich von mindestens 800 Byte zu finden. Wir gehen davon aus, dass ein solcher Bereich ab der Speicherzelle 2000 gefunden wird und der Relocating-Loader das gebundene Modul dorthin lädt. Nach dem Laden sind alle Bezüge zwischen den Modulen, die noch offen waren, aufgelöst. Das Platzierungsergebnis ist in Abbildung 10.8 dargestellt.

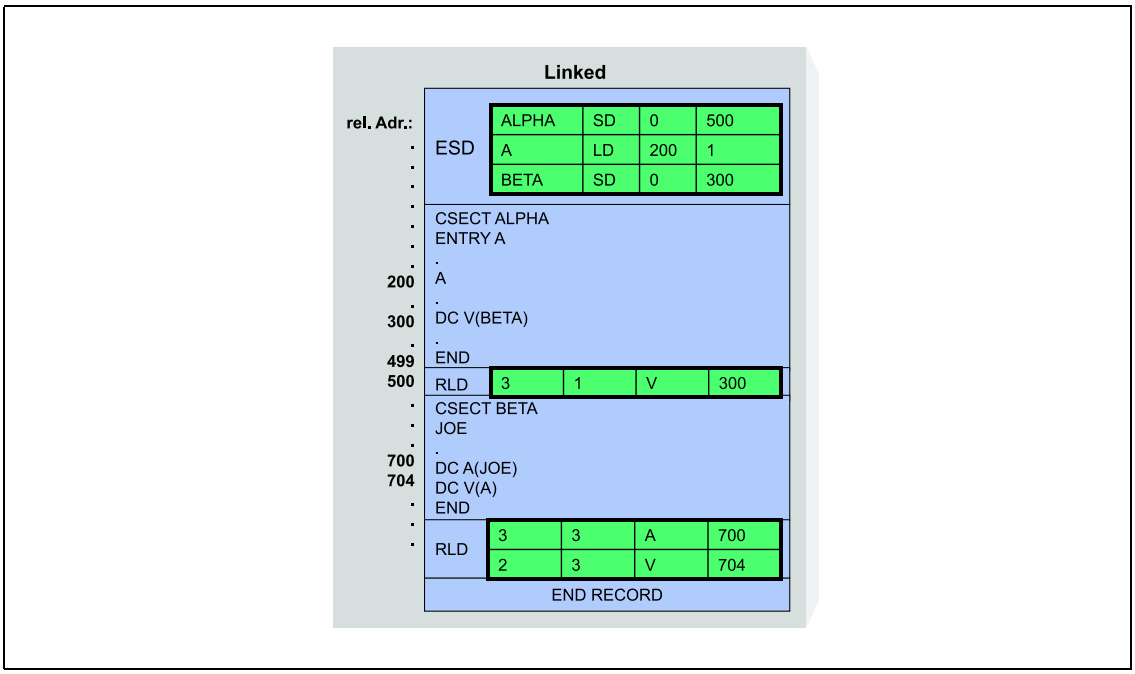


Abbildung 10.7: Gebundenes Modul

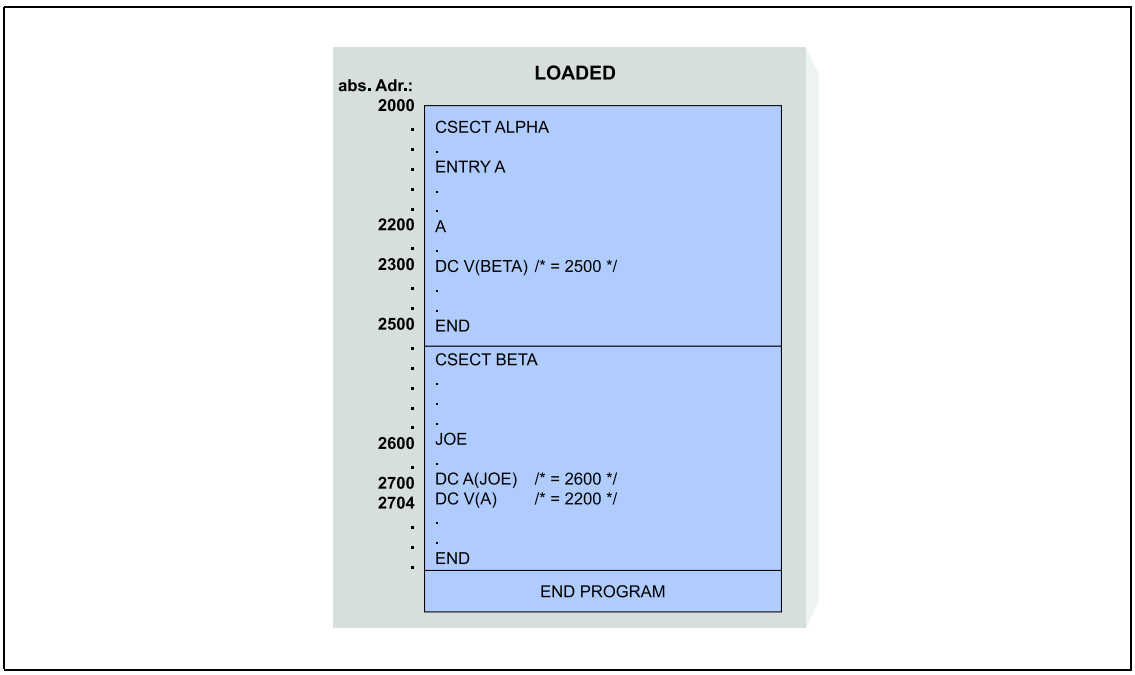


Abbildung 10.8: Modul mit absoluten Adressen nach dem Laden in Speicherbereich ab Adresse 2000

KAPITEL 11

Schutz und Sicherheit

11.1 Schutzmechanismen

Nicht jeder Benutzer oder Prozess, darf in einem Multiprogramming-System uneingeschränkt über alle Ressourcen eines Rechners verfügen. So müssen z.B. Dateien anderer Benutzer, deren Speicherbereiche etc. vor unerlaubtem Zugriff geschützt werden. Solche Maßnahmen können mehrere Gründe haben. Daten könnten z.B. unerlaubt gelesen, geändert oder gelöscht werden oder ein Prozess könnte sich Prioritäten oder Speicherbereiche aneignen, die ihm gar nicht zustehen. Man unterscheidet im Allgemeinen zwischen der Politik des Schutzes (Strategie, nach der Rechte usw. verteilt werden) und den Mechanismen, mit denen diese Politik in die Tat umgesetzt wird. Da die Politik je nach System und Benutzer sehr unterschiedlich sein kann, sollen hier nur Mechanismen betrachtet werden.

11.1.1 Schutzbereiche

Abstrakt kann ein Computer als eine Menge von Prozessen und Objekten definiert werden. Prozesse wurden bereits vorgestellt. Objekte sind die Teile des Rechners, die von Prozessen genutzt werden können. Dazu gehören zum einen Hardwarekomponenten, z.B. Speicher, CPU, Festplatte, andererseits aber auch Software wie z.B. Programme oder andere Dateien. Jedes Objekt ist durch einen eindeutigen Namen gekennzeichnet, und auf ihm können objektabhängige Operationen ausgeführt werden. Beispielsweise kann ein Speicher an einer bestimmten Stelle gelesen oder ein Programm gestartet werden. Damit ein Prozess keine unerlaubten Operationen auf einem Objekt durchführen kann, muss das System Kenntnis darüber haben, was ein Prozess darf und was nicht. Dazu wird jedem Prozess (oder Benutzer oder beiden) ein so genannter Schutzbereich zugeordnet. Ein **Schutzbereich** ist eine Menge von geordneten Paaren, bestehend aus einem Objektnamen und einer Menge von Rechten, gewisse Operationen ausführen zu dürfen. Ein Element des Schutzbereichs könnte z.B. das Paar `(Datei D, {lesen, schreiben})` sein. Um die Anzahl der so verwalteten Daten zu verringern, können Teile von Schutzbereichen oder auch ganze Schutzbereiche von mehreren Prozessen gemeinsam genutzt werden (siehe Abbildung 11.1).

Zuordnungen von Schutzbereichen zu Prozessen können statisch oder dynamisch sein. Im statischen Fall werden sämtliche Elemente des Schutzbereichs bei der Erzeugung eines neuen Prozesses für seine gesamte Lebensdauer festgelegt. Bei der dynamischen Vergabe ist eine Änderung des Schutzbereichs zur Laufzeit eines Prozesses möglich. Beide Lösungen haben gewisse Nachteile. Die statische Zuteilung von Rechten erfordert eine Berücksich-

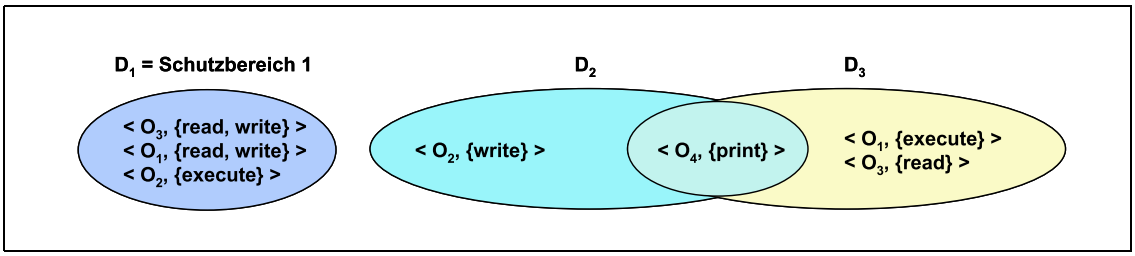


Abbildung 11.1: Schutzbereiche

tigung aller möglichen Operationen auf allen möglichen Objekten, auch wenn diese von dem Prozess gar nicht beansprucht werden (man denke an ein großes Dateisystem, bei dem für jeden Prozess und jede Datei zuerst einmal Zugriffsrechte vergeben werden). Eine dynamische Zuordnung könnte (als Extrembeispiel) eine Vergabe von Rechten nur für den Zeitraum einer Objekt-Operation ermöglichen. In jedem Fall ist die dynamische Methode jedoch wesentlich aufwendiger zu implementieren.

11.1.2 Zugriffsmatrizen

Eine Möglichkeit, Schutzbereiche zu beschreiben, sind **Zugriffsmatrizen**. Jede Zeile einer Zugriffsmatrix entspricht einem Schutzbereich. Die Spalten der Matrix werden den zu schützenden Objekten zugeordnet. Jedes Element der Matrix enthält eine Menge von Rechten.

Objekt Schutzbereich	Datei1	Datei2	Datei3	Drucker
S1	lesen		lesen	ducken
S2		ausführen	lesen	drucken
S3	lesen, schreiben		lesen	
S4		ausführen	lesen, schreiben	drucken

Diese Art der Darstellung kann sowohl für statische als auch für dynamische Schutzbereiche benutzt werden. Die Zugriffsmatrix kann als Objekt in sich selbst enthalten sein. Unter gewissen Umständen kann es für einen Prozess wünschenswert sein, seinen Schutzbereich zu wechseln. Dazu ist nur eine einfache Erweiterung der Zugriffsmatrix notwendig. An das Ende der Matrix werden sämtliche Schutzbereiche angehängt. Das Recht, eine Operation `switch` auszuführen, bedeutet dann, dass ein Prozess aus dem aktuellen Schutzbereich in den Objekt-Schutzbereich wechseln darf.

Objekt Schutzbereich	...	Drucker	S1	S2	S3	S4
S1	...	drucken				
S2	...	drucken	switch		switch	
S3	...			switch		
S4	...	drucken				

Beispielsweise kann in dem angegebenen Beispiel ein Prozess aus S2 in S1 wechseln, nicht jedoch umgekehrt. Elemente der Matrix können noch zusätzliche Werte enthalten, die Änderungen der Zugriffsmatrix erlauben.

- Ist eine Operation eines Schutzbereichs mit Kopierrecht ausgezeichnet, so kann ein Prozess dieses Schutzbereichs die Operation jedem anderen Schutzbereich zugänglich machen (Beispiel: Wäre die Operation drucken für S4 mit Kopierrecht gekennzeichnet, so könnte ein Prozess aus S4 dem Bereich S3 Druckrechte zuordnen). Diese Weitergabe von Rechten kann mehrere Ausprägungen haben: Die eigenen Rechte können gelöscht werden oder nicht. Außerdem wäre eine Übergabe des Kopierrechts denkbar.
- Hat ein Element der Matrix als Zusatz den Wert Eigentümer, so dürfen beliebige Rechte in der entsprechenden Spalte gesetzt werden.
- Die Angabe `Kontrolle` ist nur in Spalten der Objekte vom Typ Schutzbereich möglich. Ist dieser Wert beispielweise für S2 und Objekt S3 gesetzt, so kann ein Prozess aus S2 jedes Matrixelement der Zeile S3 löschen, d.h. anderen Prozessen unter Umständen Rechte entziehen.

Beispiele verschiedener Zugriffsmatrizen sind in Abbildung 11.2 dargestellt.

11.1.3 Implementierung von Zugriffsmatrizen

Die einfachste Möglichkeit, Zugriffsmatrizen zu realisieren, besteht darin, nur eine einzige Matrix in einer globalen Tabelle zu speichern. Die Tabelle besteht aus geordneten Einträgen der Form `(Schutzbereich, Objekt, Rechte)`. Immer wenn aus einem Schutzbereich S eine Operation m auf ein Objekt O aufgerufen wird, kontrolliert das System, ob ein Eintrag (S, O, R) existiert, bei dem m in R enthalten ist. Ist dies der Fall, wird m erlaubt, ansonsten nicht.

Die Zugriffsmatrix kann jedoch auch durch Zugriffslisten für Objekte implementiert sein. Jedem Objekt werden in einer geordneten Liste Tupel der Form (Schutzbereich, Rechte) angehängt (siehe Abbildung 11.3). Der Vorteil dieser Darstellungsart liegt insbesondere darin, dass Schutzbereiche ohne Rechte nicht in die Liste übernommen werden. In einem zusätzlichen Element **Default-Rechte**, können zusätzlich Rechte gespeichert werden, die für alle Schutzbereiche gelten. Wird diese Menge zuerst geprüft, kann die Suche nach dem Tupel (S, R) in der Liste evtl. komplett eingespart werden (Effizienzsteigerung).

In umgekehrter Weise können auch **Capability-Listen** für Schutzbereiche benutzt werden. Diese enthalten die Objekte zusammen mit den erlaubten Operationen. Analog zu den Zugriffslisten für Objekte können Einträge ohne zulässige Operationen weggelassen werden. Objekte werden in der Liste durch ihre physikalischen Namen bzw. Adressen repräsentiert, der so genannten Capability (siehe Abbildung 11.4). Ein Operationsaufruf enthält diese Capability des Objekts als Parameter und wird immer erlaubt, da die Capability nur bekannt ist, wenn das entsprechende Recht vorhanden ist. Da den Capabilities eine wichtige Rolle zukommt, müssen sie besonders geschützt werden. Insbesondere dürfen sie nur vom Betriebssystem geändert werden.

Ein Kompromiss zwischen den beiden zuletzt vorgestellten Verfahren stellt der **Schloss-Schlüssel-Mechanismus** dar. Jedes Objekt hat eine Liste eindeutiger Bitmuster (Schlösser). Listen von Bitmustern gehören ebenfalls zu jedem Schutzbereich (Schlüssel). Ein

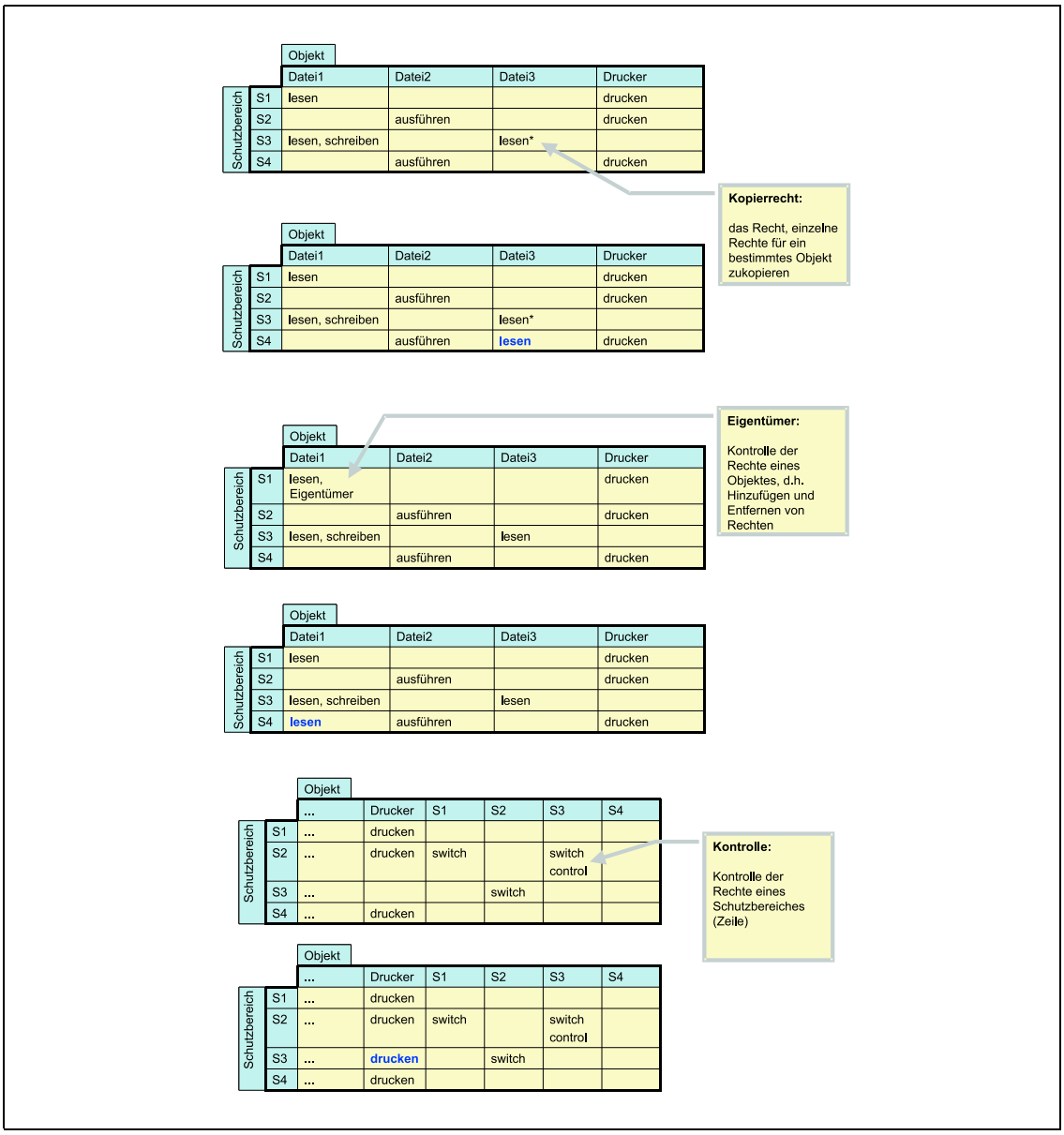


Abbildung 11.2: Beispiele von Zugriffsmatrizen

Prozess darf eine Operation ausführen, wenn sein Schutzbereich den Schlüssel zu einem Schloss des Objekts enthält (siehe Abbildung 11.5).

11.1.4 Entzug von Zugriffsrechten

Sollen zu einem Objekt gehörende Rechte entzogen werden, stellen sich einige Fragen:

- Sollen die Rechte sofort oder verzögert gelöscht werden?
- Betrifft der Entzug alle Schutzbereiche oder nur einen Teil der Schutzbereiche? Können für ein Objekt Teile der Rechte entfernt werden oder nur alle gemeinsam?
- Ist der Entzug dauerhaft oder nur zeitweise?

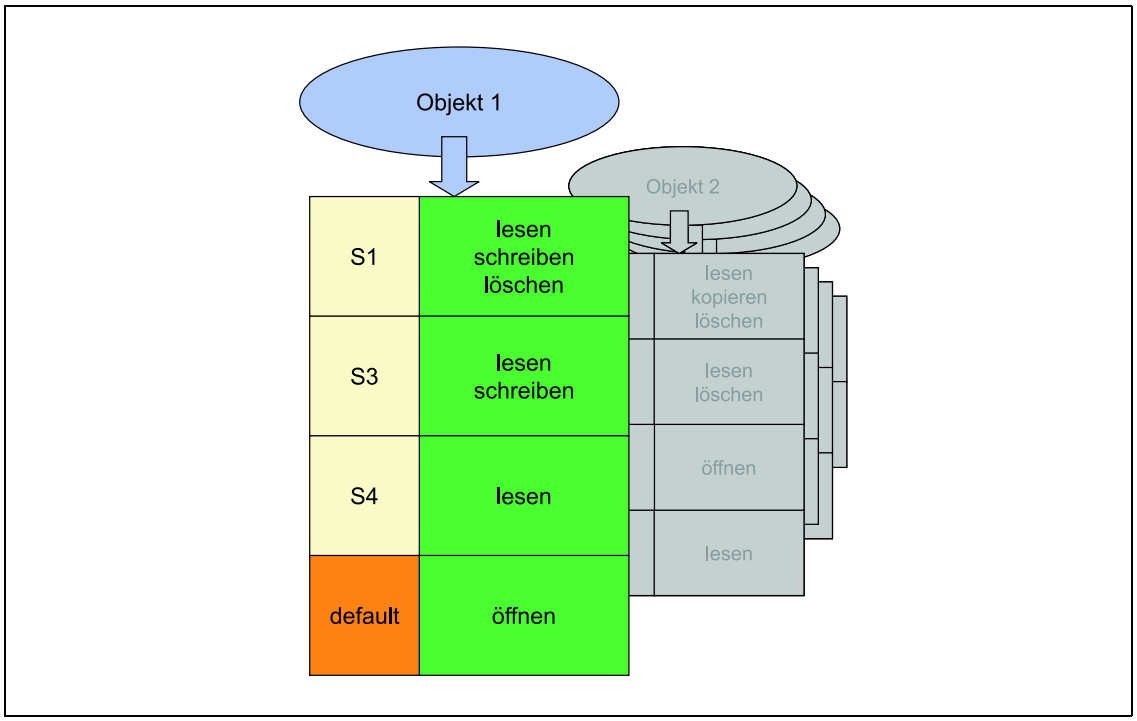


Abbildung 11.3: Implementierung von Zugriffsmatrizen mittels Zugriffslisten

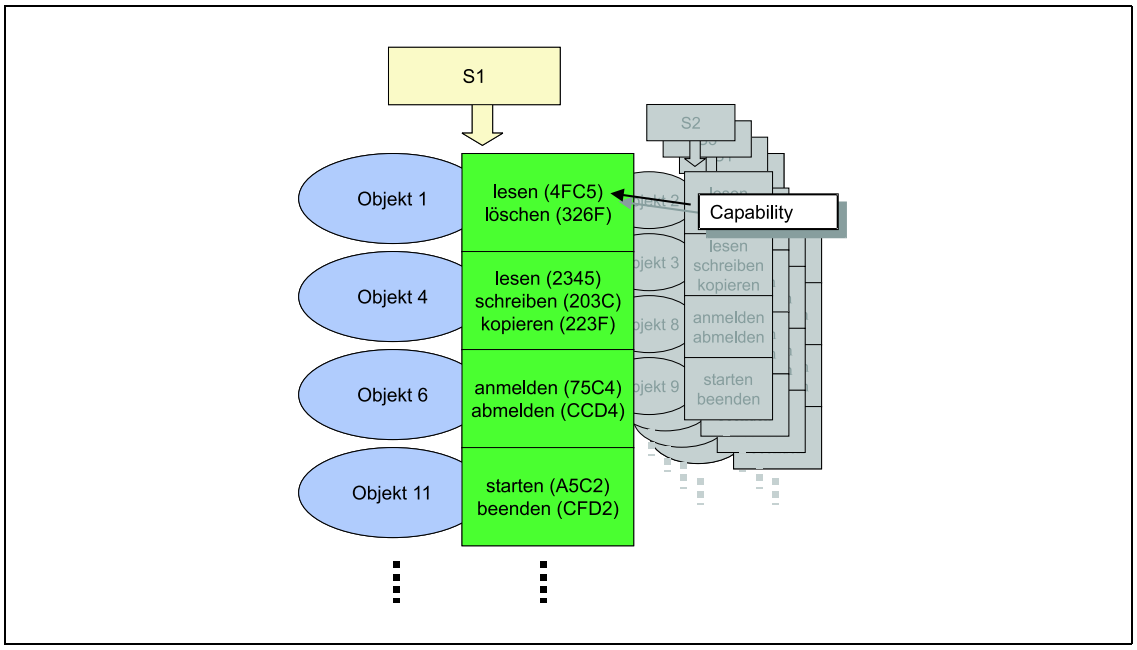


Abbildung 11.4: Implementierung von Zugriffsmatrizen mittels Capability-Listen

Sind diese Fragen geklärt, muss ein Mechanismus entwickelt werden, der eine entsprechende Entfernung durchführt. In Zugriffslisten für Objekte ist die Entfernung der Rechte einfach: Die Liste des entsprechenden Objekts wird durchsucht, und die Rechte werden gelöscht. In Capability-Listen liegt das Problem darin, dass Objekte nicht nur einfach, sondern in mehreren Listen auftauchen können. Man muss also alle Capabilities finden, um die Rechte zu entziehen. Zur Vereinfachung gibt es verschiedene Möglichkeiten:

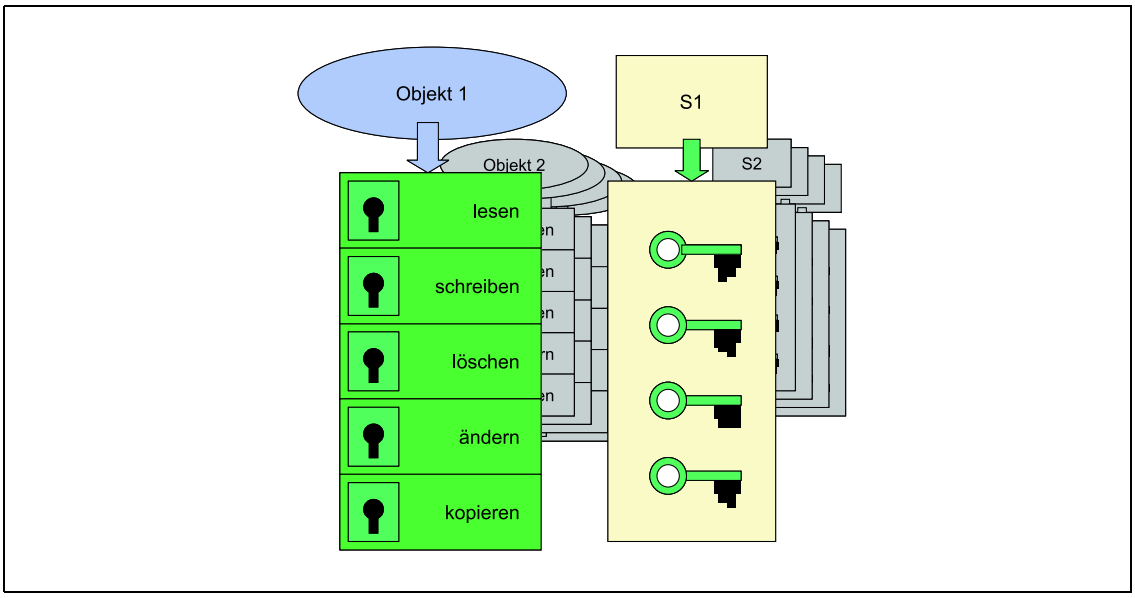


Abbildung 11.5: Implementierung von Zugriffsmatrizen mittels Schlüssel-Schloss-Mechanismus

- Periodisch werden alle Capabilities gelöscht und bei einem Zugriff neu gebildet.
- Für jedes Objekt gibt es Zeiger, die auf zugehörige Capabilities verweisen.
- Capabilities sprechen ein Objekt nicht mehr direkt, sondern über ein gemeinsames, zwischengeschaltetes Element an. Das Löschen dieses Elements entspricht dem Löschen aller Capabilities, da bei jedem Zugriff auf das Objekt ein Zeiger-Fehler auftritt.
- Jeder Capability wird bei ihrer Erzeugung der Schlüssel des Objekts übergeben (Bitmuster). Um erfolgreich auf das Objekt zugreifen zu können, müssen die Schlüssel von Objekt und Capability übereinstimmen. Sollen die Rechte entzogen werden, so wird einfach der Schlüssel des Objekts geändert.

11.2 Das Problem der Sicherheit

Die angesprochen Schutzmechanismen bewahren Daten und Ressourcen vor unberechtigtem Zugriff von Prozessen. Sie schützen aber keineswegs vor unberechtigtem Zugriff durch Menschen. Ein Benutzer, der beispielsweise das Passwort des Systemverwalters kennt, kann sich ohne Probleme als diesen ausgeben und evtl. beträchtlichen Schaden anrichten. Auch das Übertragen von vertraulichen Daten über ungesicherte Netze (wie z.B. das Internet) kann gefährlich sein, wenn die Leitung abgehört wird.

11.2.1 Authentifizierung

Wie kann ein System nun kontrollieren, wer aktuell den Rechner nutzt? Eine Möglichkeit besteht darin, zu Beginn die **Authentizität** eines Benutzers zu prüfen. Dies geschieht im Allgemeinen über ein Passwort. Es können jedoch auch andere Verfahren verwendet werden, wie z.B. Code-Karten oder (noch komplizierter) ein Vergleich von Fingerabdrücken. **Passwörter** bieten, sofern sie geeignet gewählt (also nicht unbedingt den Namen der

Freundin, dafür aber möglichst lang und mit Sonderzeichen) und gehandhabt (sprich nicht mit einem Aufkleber an der Tastatur vermerkt sind) werden, eine recht gute Möglichkeit zur Authentifizierung. Dazu benutzt der Rechner eine Funktion f, die aus einer Zeichenkette x sehr schnell einen Funktionswert $f(x)$ berechnen kann. Die umgekehrte Berechnung, d.h. von einem Funktionswert $f(x)$ auf x zu schließen, sollte praktisch unmöglich sein. Für jeden Benutzer speichert das System nur den Funktionswert seines Passworts und vergleicht diesen Wert bei jeder Authentifizierung mit dem Wert, der aus der Eingabe des Passworts berechnet wird. Eine Übereinstimmung besagt, dass der Benutzer erfolgreich identifiziert wurde. In Netzwerken, bei denen Passwörter teilweise unverschlüsselt übertragen werden, ist diese Sicherheit jedoch nicht mehr gegeben. Hier werden kryptographische Verfahren notwendig, auf die später noch etwas genauer eingegangen wird.

11.2.2 Viren und ähnliches Gewürm

In vielen Systemen stehen einige Programme mehreren Benutzern gemeinsam zur Verfügung, z.B. Editoren, Compiler etc. Diese Programme können als **Trojanische Pferde** missbraucht werden. Ein Benutzer, der ein gemeinsames Programm aufruft, öffnet diesem praktisch die Tore zu allen seinen Rechten. Ein Compiler könnte z.B. eine Routine enthalten, die alle eigentlich nur von dem Benutzer lesbaren Dateien in ein anderes, für alle sichtbares Verzeichnis kopiert. Ein anderes Problem besteht darin, dass Programmierer eines Betriebssystems so genannte **Trap Doors** in den Code einbauen können. Zum Beispiel könnte ein bestimmtes Passwort einem Benutzer damit Zugriff zu allen Daten ermöglichen. Dies ist besonders dann gefährlich, wenn es um sensitive Daten geht wie z.B. Kontostände in den Rechnern einer Bank.

Innerhalb von Computer-Netzwerken können **Würmer** eine Gefahr darstellen. Würmer sind eigenständige Programme, die auf verschiedene Art und Weisen versuchen, Zugriff auf andere Rechner zu bekommen, um dort eine Kopie zu starten. Da in manchen Betriebssystemen beispielsweise die Möglichkeit besteht, die Abfragen von Passwörtern bei entferntem Einloggen abzuschalten, hat ein solches Programm gute Chancen, sich im Netz auszubreiten und Schaden anzurichten.

Eine andere Form sich ausbreitender Gefahren sind Viren. Ein **Virus** ist kein eigenständiges, sondern nur ein Teil eines infizierten Programms. Die Verteilung dieser Programmteile erfolgt über öffentlich zugängliche Software-Datenbanken oder einfach durch den Austausch von Disketten. Vor all diesen Problemen kann man sich nie perfekt schützen. Ein vorsichtiger Umgang mit der benutzten Software kann jedoch erheblich zur Verringerung beitragen. Außerdem kann das Betriebssystem durch **Monitoring** mögliche Gefahrenquellen ausmachen. Beispielsweise kann das System von Zeit zu Zeit nach bekannten Viren durchsucht oder wichtige Verzeichnisse auf ihre Konsistenz überprüft werden.

11.2.3 Kryptographie

Die **Kryptographie** beschäftigt sich mit der Verschlüsselung von Informationen. Unter Benutzung kryptographischer Verfahren werden Klartexte so verändert, dass die **chiffrierten** (kodierten) Daten für unberechtigte Leser nicht zu entziffern sind. Zum Lesen berechtigte Benutzer besitzen hingegen einen **Schlüssel**, mit dem die chiffrierte Information **dechiffriert** (dekodiert) werden kann. Im Allgemeinen werden für den hier beschriebenen Vorgang drei Dinge benötigt: ein Kodier-Algorithmus K, ein Dekodier-Algorithmus D und ein geheimer Schlüssel s. Ein für die Übertragung einer Nachricht m geeignetes Kryptographie-Verfahren muss folgenden Bedingungen genügen:

1. $D(s, K(s, m)) = m$.
2. D und K sind effizient berechenbar.
3. Die Sicherheit ist nur von der Geheimhaltung des Schlüssels s, nicht jedoch von den Algorithmen D und K abhängig.

In unsicheren Netzwerken benutzt man zur Übertragung der Daten häufig einen Mechanismus, der **öffentliche** und **private Schlüssel** verwendet (siehe Abbildung 11.6). Eine mit dem (allen bekannten) öffentlichen Schlüssel kodierte Nachricht kann nur mit dem entsprechenden privaten Schlüssel dekodiert werden, der somit nur dem Empfänger bekannt sein darf. Um eine Nachricht kodiert zu verschicken, muss also nur ein öffentlicher Schlüssel des Empfängers angefordert werden. Das Problem dabei ist: Wie findet man geeignete Schlüsselpaare? Ein bekanntes Verfahren, das Ergebnisse der Zahlentheorie verwendet, ist der nach seinen Erfindern Rivest, Shamir und Adleman benannte **RSA-Algorithmus**:

1. Wähle zwei große Primzahlen p und q ($> 10^{100}$).
2. Berechne $p \cdot q$ sowie $z = (p-1) \cdot (q-1)$.
3. Wähle ein $d < n$ teilerfremd zu z.
4. Finde ein $e < n$ mit $e \cdot d = 1 \bmod z$.

Dann ist das Paar (e, n) der öffentliche und (d, n) der private Schlüssel. Kodiert wird eine Nachricht m mit $c = m^e \bmod n$, dekodiert mit $m = c^d \bmod n$. Um diese Kodierung zu knacken, müsste n in seine Primfaktoren zerlegt werden, was mit den heutigen Methoden (Mathematiker + Computer) und bei genügend großem n praktisch unmöglich ist.

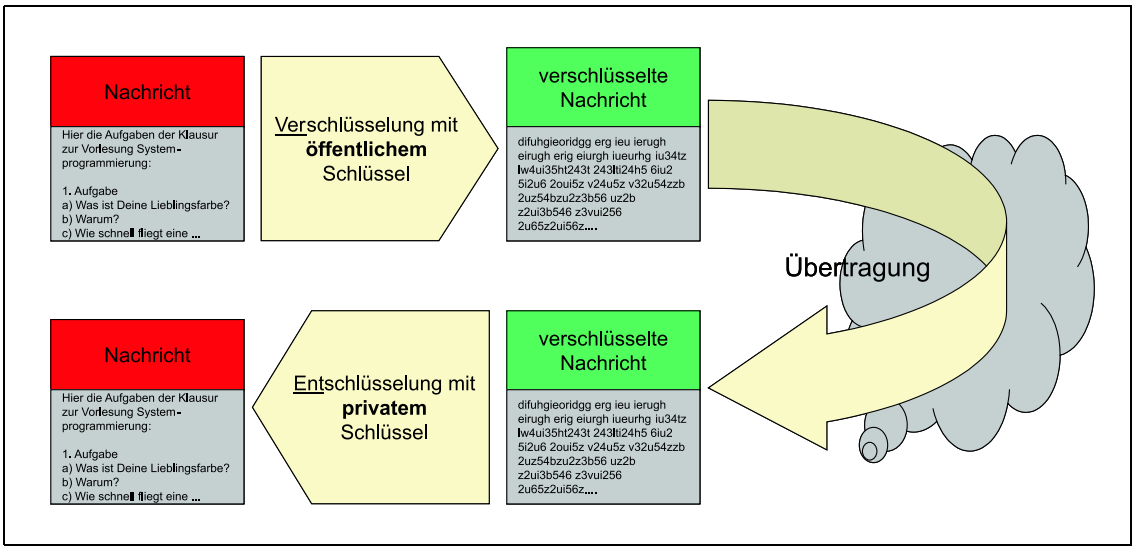


Abbildung 11.6: Kryptographie

KAPITEL 12

Verteilte Systeme

Betrachtet man nicht einen isolierten Rechner, sondern gestaltet dessen Möglichkeiten durch die Verbindung zu anderen Rechnern komfortabler, so entsteht ein **Verteiltes System**. Dieser Begriff ist inzwischen Gegenstand zahlreicher Bücher. So verwendet etwa Tanenbaum in seinem Werk *Moderne Betriebssysteme* etwa die Hälfte des Buches zur Beschreibung Verteilter Systeme; das Nachfolgebuch des gleichen Autors bekam dann sogar den Titel *Verteilte Betriebssysteme*. Auch wenn man sich real existierende Rechnernetze und Informationssysteme anschaut, ist die wachsende Bedeutung der Verteilten Systeme nicht zu leugnen. Allgemein kann man sagen, dass sich netzspezifische Fächer oder auch die Vorlesung *Datenkommunikation* mit den Schichten 1 - 4 (bezogen auf das in Kapitel 1 eingeführte OSI-Modell) beschäftigen, während unter dem Stichwort *Verteilte Systeme* anwendungsnah die Aufgaben der oberen Schichten $5-7$ des OSI-Modells untersucht werden.

12.1 Grundlagen Verteilter Systeme

Innerhalb dieses Abschnitts soll auf die Grundlagen Verteilter Systeme eingegangen werden. Ausgangspunkt dabei ist eine Begriffsbestimmung. Es schließt sich eine Diskussion an, ob Verteilte Systeme sinnvoll im Einsatz sind. Schließlich wird auf ein System, das für einen Zusammenschluss einzelner Rechner zu einem Verteilten System notwendig sind, eingegangen: die Verteilungsplattform.

12.1.1 Funktionalität und Anforderungen

In der Literatur gibt es eine Reihe verschiedener Ansätze zur Klärung dessen, was man unter einem Verteilten System genau verstehen soll.

DEFINITION

Verteiltes System

Ein **Verteiltes System** (siehe Abbildung 12.1) ist demnach ein System mit räumlich verteilten Komponenten, die keinen gemeinsamen Speicher benutzen und einer dezentralen Administration unterstellt sind. Zur Ausführung gemeinsamer Ziele ist eine Kooperation der Komponenten möglich. Werden von diesen Komponenten Dienste angeboten und angebotene Dienste genutzt, so entsteht ein **Client/Server-System**, im Falle einer zusätzlichen zentralen Dienstvermittlung ein so genanntes Dienstvermittlungs- oder **Tradingsystem**.

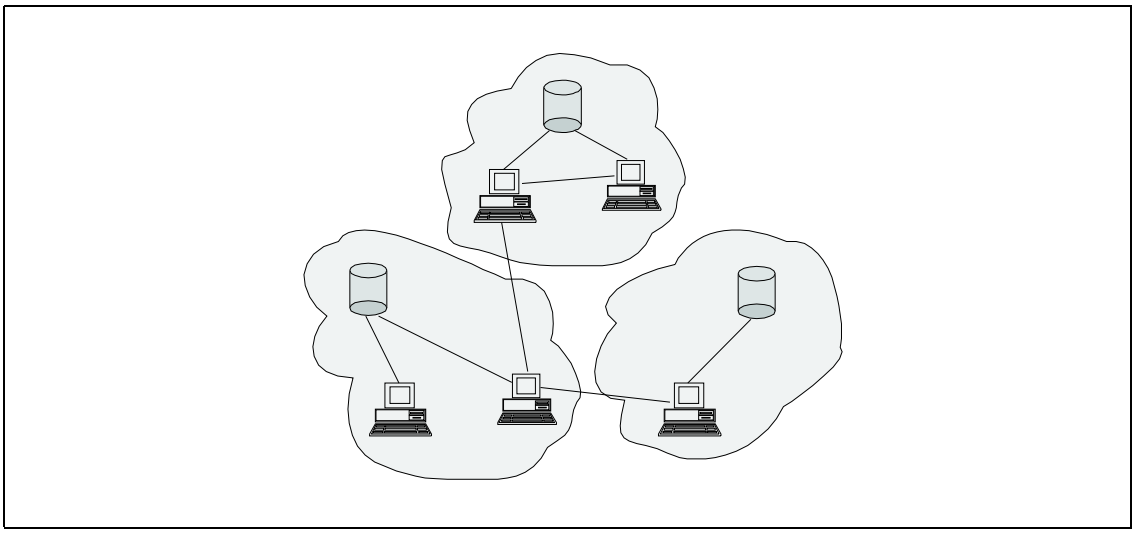


Abbildung 12.1: Verteiltes System

Das Konzept der Verteilten Systeme ist relativ jung. Erst in der Mitte der 80er Jahre wurden die Voraussetzungen dafür geschaffen, dieses Prinzip einführen zu können. Als grundlegend für die Entwicklung bzw. Installation eines Verteilten Systems muss man dabei festhalten:

Leistungsexplosion bei Halbleiterchips: Die Rechenleistung bei Mikroprozessoren hat sich im letzten Jahrzehnt ca. alle zwei Jahre verdoppelt, die Kapazität von Halbleiterspeichern alle drei Jahre vervierfacht. Stetig wachsende Leistung bei schrumpfenden Preisen und Abmessungen bilden die Grundlage dafür, dass immer mehr Rechner immer komplexere Software ausführen können.

Schnelle Rechnernetze: Die Bereitstellung schneller lokaler Datennetze senkt die Zugriffszeiten auf externe Rechner und bildet so die ökonomische Voraussetzung dafür, Computer (PCs wie Workstations) miteinander zu verbinden. Die Einführung der Ethernet-Technik in den siebziger Jahren kann als Wegbereiter für verteilte Softwaresysteme angesehen werden.

Konzepte der Softwaretechnik: In den letzten drei Jahrzehnten waren erhebliche Fortschritte im Bereich der Softwaretechnik zu verzeichnen. Die Akzeptanz von programmiersprachlichen Konzepten wie Prozedur, Modul und Schnittstelle schuf die Voraussetzungen für die grundlegenden Mechanismen Verteilter Systeme. Konsequenzen waren der **Remote-Procedure-Call** (RPC) und die objektorientierte Modellierung Verteilter Systeme.

Autonomie der Rechnerorganisation: Die Abkehr von streng hierarchisch aufgebauten Organisationsformen in Unternehmen führt ganz allgemein zu einer Dezentralisierung und schafft flache Führungsstrukturen. Dadurch eröffnen sich weite Anwendungsfelder für Verteilte Systeme.

Verteilte Systeme besitzen eine Reihe von Vor- und Nachteilen. Auch wenn es beim heutigen Stand der Technik möglich ist, Verteilte Systeme zu installieren, so ist doch zu überlegen, warum beispielsweise ein bestehendes zentrales System durch ein Verteiltes System ersetzt werden sollte.

12.1.2 Vor- und Nachteile Verteilter Systeme

Das noch vor einigen Jahren typische Rechenzentrum eines Großunternehmens ist heute kaum noch anzutreffen. Stattdessen verfügen die meisten Unternehmen über Verteilte Systeme. Diese Beobachtung aus der Wirtschaft lässt bereits vermuten, dass letztlich wohl doch die Vorteile überwiegen. Daher wollen wir im Folgenden zunächst auf die wichtigsten Vorteile von Verteilten Systemen eingehen:

Stetige Kapazitätsanpassung: Verteilte Systeme ermöglichen die stetige Anpassung der Größe eines Systems. Den neu hinzukommenden Anforderungen an das Computersystem eines expandierenden Unternehmens kann durch Erweiterungen bestehender Komponenten zeitgemäß entsprochen werden.

Integrierbarkeit bestehender Lösungen: Existierende Systeme können von neu hinzukommenden Systemkomponenten genutzt werden, ohne dass ein System gleicher Funktionaliät neu entwickelt werden muss.

Risikominimierung: Eine sukzessive Systemerweiterung minimiert das Risiko der Überlastung einzelner Systemkomponenten, da hierbei stets auf die gleichmäßige Auslastung sowohl bestehender als auch neu hinzukommender Module geachtet werden kann.

Flexibilität und Anpassbarkeit: Die überschaubare, organisatorische Verwaltung der Kapazität eines Verteilten Systems erlaubt kostengünstige Realisierungen. Das System ist flexibel und anpassbar.

Autonomie: Der Eigentümer einer Ressource hat die Möglichkeit, das Management dieser Komponente selbst zu übernehmen. In jedem Fall steht es ihm frei, bei Bedarf einzugreifen, um seine eigenen Interessen wahrzunehmen. Ferner sind die einzelnen Bestandteile eines Verteilten Systems weitestgehend autonom. Im Falle eines Fehlers oder sogar Ausfalls einer Systemkomponente können die übrigen Einheiten im Idealfall unbeeinflusst weiterarbeiten und ggf. den Störfall überbrücken.

Es gibt zahlreiche weitere Vorteile Verteilter Systeme. Diese reichen von besserer Bearbeitung einzelner Prozesse durch parallele Abarbeitung bzw. Nebenläufigkeit bis hin zu Möglichkeiten, Mobilität in Verteilten Systeme zu erhalten.

Bei der Diskussion der Nachteile - über die man sich auch streiten kann, da sie vielleicht auch nur eine Frage der Forschung und Entwicklung auf diesem Gebiet sind - seien folgende Punkte genannt:

Softwaredefizit: Für den Zeitraum des Übergangs zu Verteilten Systemen droht ein Softwaredefizit. Die Realisierung eines Verteilten Systems erfordert komplexere Softwarelösungen als die eines zentralen Systems. In dieser Hinsicht stehen die Verteilten Systeme erst am Anfang der Konzeptentwicklung.

Sicherheitsbedenken: Die neu hinzukommenden Netzwerkkomponenten können vollkommen neuartige Fehler verursachen. Unabhängig davon sind Verteilte Systeme auch aus Sicht des Datenschutzes bedenklich, da vernetzte Daten generell einen einfacheren Zugriff ermöglichen, als dies bei separater Datenhaltung der Fall ist.

Leistungsfähigkeit: Ein Prozessorpool bedingt im Mittel eine kleinere Wartezeit bis zur Abarbeitung einer Aufgabe, als dies bei mehreren einzelnen Komponenten mit insgesamt der gleichen Prozessorleistung der Fall ist. Um annähernd gleiche Größen zu erhalten, ist ein komfortables Leistungsmanagement bei Verteilten Systemen erforderlich.

12.1.3 Verteilungsplattformen

Verteilte Systeme zeichnen sich durch eine Reihe von Eigenschaften aus, die zentrale Systeme nicht benötigen bzw. über die sie gar nicht erst verfügen. Eine der wichtigsten Eigenschaften ist in diesem Zusammenhang die so genannte Transparenz.

Transparenz ist eine zentrale Forderung, die sich aus dem Wunsch ergibt, den Umgang mit verteilten Anwendungen so weit wie möglich zu erleichtern. Sie umfasst das Verbergen von Implementierungsdetails und die so genannte Verteilungstransparenz. Diese verbirgt ihrerseits die Komplexität eines Verteilten Systems. Dabei werden interne Vorgänge durch so genannte Transparenzfunktionen vor dem Betrachter verborgen und der Nutzer entlastet.

Es gibt viele Ausprägungen von **Verteilungstransparenz**. An dieser Stelle seien nur die wichtigsten Stichworte genannt:

- Zugriffstransparenz,
- Ortstransparenz,
- Replikationstransparenz,
- Abarbeitungstransparenz,
- Migrationstransparenz,
- Ausfalltransparenz,
- Ressourcentransparenz,
- Verbundtransparenz und
- Gruppentransparenz.

Zur Realisierung verteilter Zugriffe bedarf es einer geeigneten Softwareinfrastruktur. Diese heißt **Verteilungsplattform** oder auch **Netzbetriebssystem** bzw. **Middleware** und dient zur Unterstützung der Interaktion zwischen den auf potenziell heterogenen Systemen ablaufenden Anwendungskomponenten. Die Verteilungsplattform wird dem lokalen Betriebssystem hinzugefügt oder übernimmt selbst Aufgaben eines Betriebssystems (siehe Abbildung 12.2).

Mittels einer solchen Verteilungsplattform lässt sich die Verteilung transparent halten, d.h. Anwendungen werden von komplexen Details interner Vorgänge abgeschirmt. Einer Verteilungsplattform können viele individuelle Systeme zugrunde liegen, auf denen sie aufsetzt. Gleichzeitig kann eine Vielzahl von Anwendungen auf die Verteilungsplattform zugreifen.

Heutzutage werden verschiedene Verteilungsplattformen eingesetzt. Die wichtigsten dieser Plattformen sind Distributed Computing Environment (**DCE**), Common Object Request Broker Architecture (**CORBA**) sowie **ANSAware**.

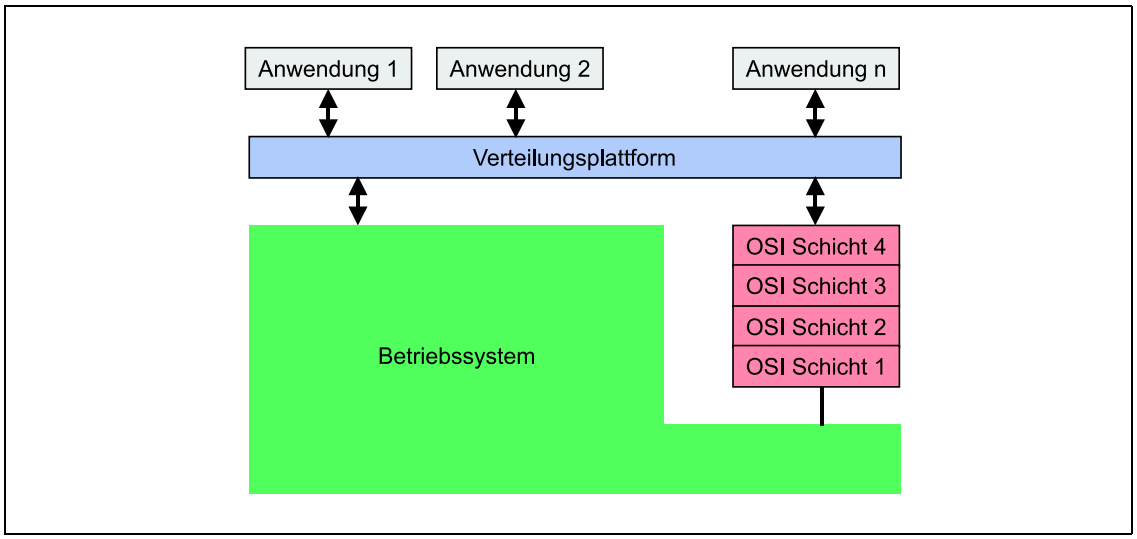


Abbildung 12.2: Die Schnittstellen einer Verteilungsplattform

12.2 Strukturen in Verteilten Systemen

In diesem Abschnitt soll auf das den Verteilten Systemen zugrunde liegende Prinzip des **Client/Server-Modells** eingegangen und damit zusammenhängende Probleme angesprochen werden.

12.2.1 Das Client/Server-Modell

Warum ein neues Modell? Nach der Vorstellung des ISO/OSI-Modells im Eingangskapitel liegt doch eigentlich die Vermutung nahe, dass damit alle mit Kommunikation zusammenhängenden Fragen angemessen behandelt werden können. Die Antwort auf die Frage liegt im Verwaltungsaufwand der sieben Schichten, der so hoch ist, dass nach einem anderen (einfacheren) Modell gesucht wurde.

Die Idee des Client/Server-Modells besteht darin, ein Betriebssystem als eine Menge kooperierender Prozesse – so genannter Server – zu strukturieren. Diese stellen den Nutzern – so genannten Clients – Dienste bereit. Auf einem Rechner können sowohl Clients als auch Server (ggf. auch gleichzeitig) laufen. Das Client/Server-Modell basiert auf einem einfachen, verbindungslosen Anfrage/Antwort-Protokoll (siehe Abbildung 12.3). Der Client sendet dabei eine Anfrage und erhält seine Antwort vom Server.

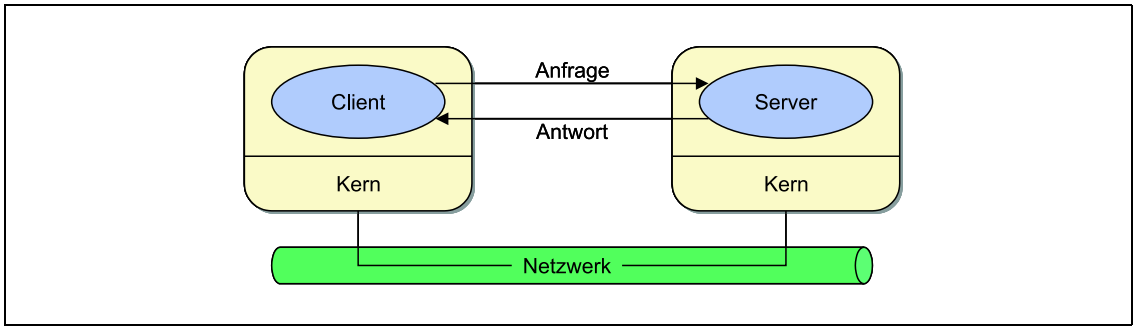


Abbildung 12.3: Das Prinzip des Client/Server-Modells

Diese einfache Methode ist sehr effizient, da der Protokollstapel klein gehalten wird. Basierend auf dieser einfachen Struktur ist es lediglich notwendig, dass die Verteilungsplattform zwei Systemaufrufe anbietet. Ein Aufruf der Syntax `send(a,&mp)` verschickt die Nachricht, die durch `mp` referenziert wird, an einen Prozess, der durch `a` identifiziert wird. Der Aufrufer wird dabei solange blockiert, bis die Nachricht vollständig verschickt ist. Demgegenüber wird der Aufrufer von `receive(a,&mp)` blockiert, bis eine Nachricht für ihn angekommen ist. Die Nachricht wird in den durch `mp` angegebenen Puffer kopiert. Der Parameter `a` gibt dabei die Adresse des Empfängers an. Bei der Realisierung eines Client/Server-Modells ist die Synchronisation der `send`- und `receive`-Aufrufe von besonderer Bedeutung. Nur durch ein geeignetes Zusammenspiel dieser beiden Operationen kann das System auch ordnungsgemäß arbeiten.

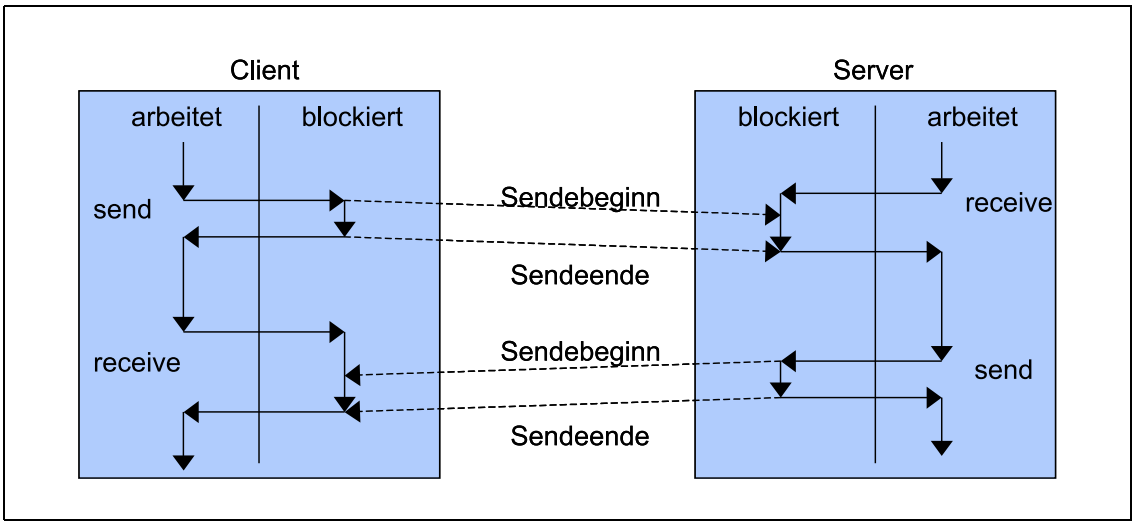


Abbildung 12.4: Die Wirkungsweise des Client/Server-Modells

Zur Veranschaulichung der Wirkungsweise ist in Abbildung 12.4 ein Client sowie ein Server dargestellt. Beide Komponenten können sich in je zwei Zuständen befinden: blockiert und nicht blockiert. Der Zustand blockiert bedeutet dabei, dass der Prozessor der Komponente nicht arbeitet, sondern auf ein Ereignis wartet, das eine Modifikation des Systemzustands bedingt. In der Regel wird dieses Ereignis bei einem `receive`-Aufruf das Ankommen einer Nachricht sein. Beim Aufruf einer `send`-Primitive erfolgt solange eine Blockierung, bis ein Nachrichtenpaket vollständig verschickt ist.

Beim Client/Server-Modell gibt es verschiedene Probleme, die genauer untersucht werden müssen. Zunächst wollen wir uns um die Adressierung kümmern.

Adressierung

Im Normalfall ist die Adresse eines anzusprechenden Servers bekannt. Die Adresse ist eine Konstante, die der Client (woher auch immer - siehe folgende Fälle) kennt, zum Beispiel kann sie in einer Datei abgelegt sein. Ist diese Voraussetzung jedoch nicht erfüllt, so ergibt sich das Adressierungsproblem (siehe Abbildung 12.5).

Die einfachste Form einer Adressierung ist die Angabe von Prozess und Zielrechner, die so genannte *machine-process-Adressierung*. Diese Alternative zur absoluten Adressierung spaltet den Namen eines Prozesses in zwei Teile auf: den Prozess und den Rechner, auf dem dieser Prozess läuft. In welcher Reihenfolge diese beiden Komponenten des Namens angegeben werden, ist Vereinbarungssache: Läuft ein Prozess mit der Nummer 4 auf einem

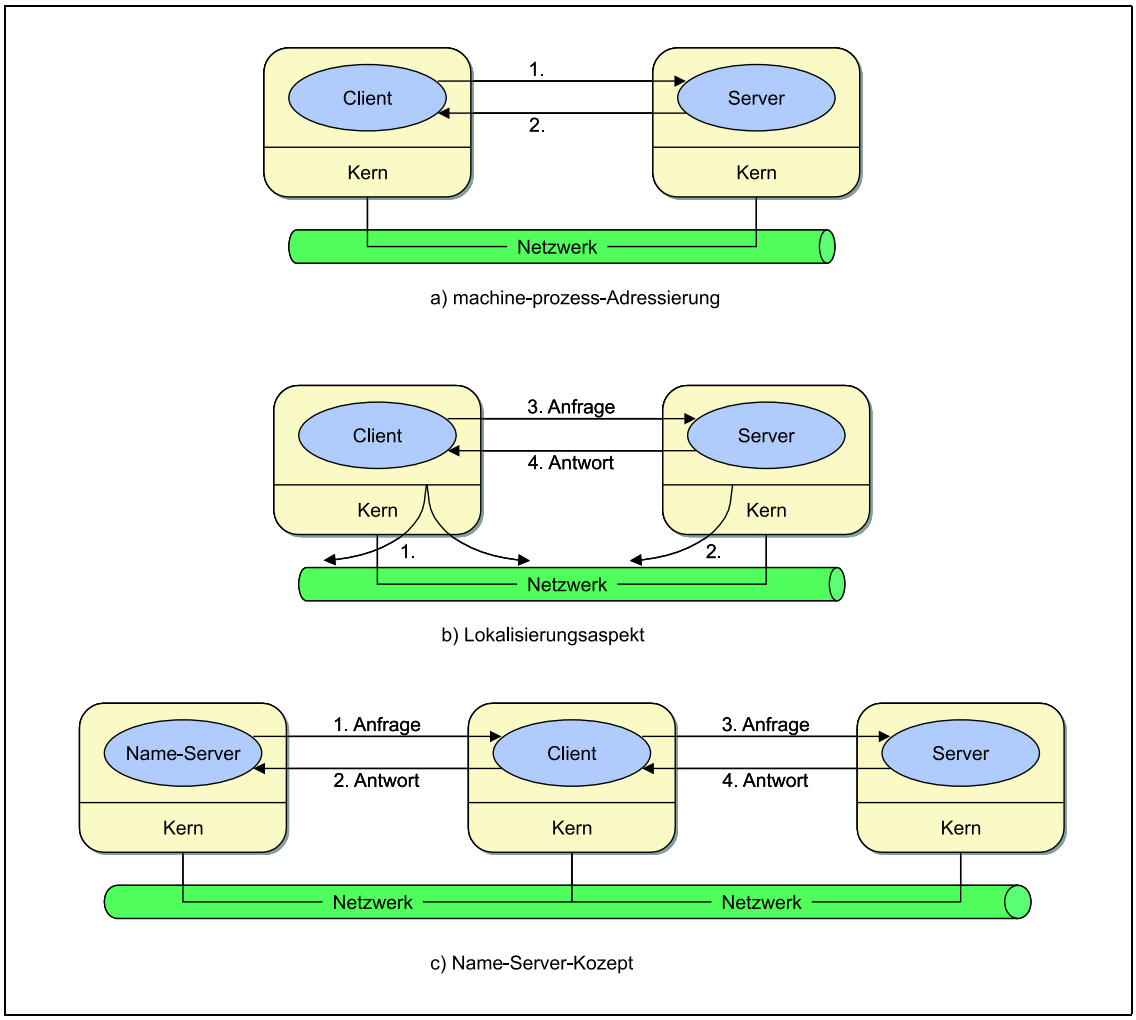


Abbildung 12.5: Adressierungsarten

Rechner 21415, so könnte als Bezeichnung gleichermaßen 4@21415 oder 21415.4 verwendet werden. Die Funktionsweise ist denkbar einfach: Nach dem Stellen einer Anfrage an einen Server erfolgt die Antwort zurück an den Client. Nachteil dieser Adressierung ist das Nichtvorhandensein von Ortstransparenz, da dem Nutzer nicht verborgen bleibt, wo er arbeitet.

Ein zweiter Ansatz sieht ein spezielles **Lokalisierungspaket** vor, das der Sender an alle anderen Rechner schickt. Das Lokalisierungspaket enthält die Adresse des Zielrechnerprozesses oder eine Bezeichnung des gesuchten Prozesses, d.h. Dienstes. Insbesondere in broadcastfähigen LANs ist diese Adressierung realisierbar. Dazu geht man in vier Schritten vor:

1. Broadcasten des Lokalisierungspakets
2. *Ich-bin-hier*-Antwort des Servers
3. Anfrage an den Server
4. Antwort des Servers.

Dieses Vorgehen sichert zwar die Ortstransparenz, verursacht in manchen Netzen durch

den Broadcastaufruf jedoch zusätzliche Last. Aus diesem Grunde gibt es eine dritte Alternative, die gerade in Verteilungsplattformen der Normalfall ist: die Einführung eines **Name-Servers** oder **Traders**.

Diese ausgesprochen komfortable Realisierung läuft prinzipiell in folgenden vier Schritten ab:

1. Anfrage des Clients an den Trader nach der Adresse eines gewünschten Servers
2. Übermittlung der gewünschten Adresse vom Trader an den Client
3. Anfrage des Clients an die benannte Adresse eines Servers
4. Antwort des Servers an den Client

Das Vorhandensein eines solchen Traders kann dem Client auch bei unvollständigen Angaben die Vermittlung eines Dienstes ermöglichen. Diese Realisierung der Dienstvermittlung erfordert jedoch einige neue Mechanismen. Abbildung 12.5 stellt diese drei Alternativen noch einmal gegenüber.

Blockierung

In Analogie zu den in Abbildung 12.4 eingeführten Mechanismen unterscheidet man zwischen blockierenden und nichtblockierenden Primitiven. Es ist die Aufgabe des Systementwicklers, zwischen diesen beiden Möglichkeiten zu unterscheiden.

Bei **blockierenden Primitiven** wird der Prozess während der Sendung einer Nachricht blockiert, d.h. Anweisungen werden erst dann weiter abgearbeitet, wenn die Nachricht vollständig abgesendet ist. Analog endet die Blockierung beim Empfangen erst, nachdem die Nachricht angekommen und kopiert ist.

Bei **nichtblockierenden Primitiven** wird die zu sendende Nachricht zunächst nur in einen Puffer des Betriebssystemkerns kopiert. Direkt im Anschluss daran, also noch vor der eigentlichen Sendung, erfolgt die Entblockierung. Der sendende Prozess kann parallel zur Nachrichtenübertragung mit seiner Ausführung fortfahren. Dieser Geschwindigkeitsvorteil wird jedoch dadurch erkauft, dass der Sender nicht weiß, wann die Übertragung beendet ist und wann er den Puffer wieder nutzen kann. Ein zusätzlicher Nachteil ist, dass ein einmal beschriebener Puffer nicht mehr verändert werden kann.

Pufferung

Auch bezüglich der Pufferung gibt es zwei Möglichkeiten, Primitive zu realisieren. Bislang wurde immer von **ungepufferten Primitiven** ausgegangen. `receive(a,&mp)` informiert den Kern seiner Maschine, dass der aufrufende Prozess die Adresse `a` abhören will und bereit ist, Nachrichten von dort zu empfangen. Einen Puffer stellt er an der Adresse `&mp` bereit. Problematisch bei dieser Herangehensweise kann sein, dass im Falle des früheren Abschickens eines `send` und späteren Aufrufens von `receive` der Kern bei einer ankommenden Nachricht nicht weiß, ob einer seiner Prozesse die Adresse dieser Nachricht benutzen wird und wohin er ggf. die Nachricht kopieren soll. In diesem Fall kann der Verlust einer Nachricht nicht ausgeschlossen werden. Auch wenn mehrere Clients dieselbe Adresse benutzen, sind Probleme nicht ausgeschlossen.

Eine alternative Implementierung sind die **puffernden Primitive**. Ein empfangender Kern speichert die eintreffenden Nachrichten für einen bestimmten Zeitraum zwischen. Wird kein passendes `receive` aufgerufen, so werden die Nachrichten nach Ablauf eines Timeouts gelöscht. Es müssen Puffer vom Kern bereitgestellt und verwaltet werden. Eine konzeptionell einfache Lösung sieht dafür die Definition einer Datenstruktur *Mailbox* vor.

Zuverlässigkeit

Client/Server-Systeme lassen sich hinsichtlich ihrer Zuverlässigkeit in drei Klassen einteilen. Die erste Klasse verfährt nach dem Prinzip der gelben Post: Abschicken, der Rest ist egal - in der Regel kommt schon alles an. Eine sicherere Möglichkeit besteht darin, dass der empfangende Kern eine Bestätigung an den Sender schickt. Und die dritte Möglichkeit geht schließlich davon aus, dass die Antwort des Servers an den Client gleichzeitig auch die Bestätigung der Anfrage ist, also nur noch der Client bestätigt, dass er die Antwort vom Server erhalten hat.

12.2.2 Der Remote-Procedure-Call

Das **Client/Server-Modell** bietet einen brauchbaren Weg, ein Verteiltes Betriebssystem zu strukturieren. Trotzdem hat es eine extreme Schwachstelle: Das Basisparadigma aller Kommunikation ist die Ein- und Ausgabe von Daten, der explizite Aufruf von Kommunikationsprimitiven wie `send` und `receive` dient jedoch zum Austausch von Daten. Wollen wir unser Ziel erreichen, verteilte Berechnungen genauso aussehen zu lassen wie zentrale Berechnungen, dann benötigen wir komfortablere Mechanismen.

Der Vorschlag von Birell und Nelson aus dem Jahre 1984 besteht darin, dass ein Programm ein Unterprogramm aufruft, welches sich auf einem anderen Rechner befindet. Diese Methode ist als *entfernter Unterprogrammaufruf* (**Remote-Procedure-Call**, RPC) bezeichnet worden. Ruft ein Rechner A ein Unterprogramm auf einem Rechner B auf, so wird der aufrufende Prozess auf A ausgesetzt (suspendiert), und die Ausführung des aufgerufenen Unterprogramms findet auf Rechner B statt. Dabei werden Parameter ausgetauscht, der Nachrichtenaustausch bleibt für den Benutzer jedoch unsichtbar. Sei `read` ein entferntes Unterprogramm. Dann ist nicht dieses `read` in der Bibliothek eines Clients enthalten, sondern die Bibliothek enthält einen so genannten Client-Stub, an den sich der Client mit seiner Anfrage wendet und von dem er letztlich auch die Antwort bekommt. Ein **Stub** wird zuweilen auch als Stellvertreterprozedur bezeichnet. Er ist ein Prozess, der die Aufgabe besitzt, aus einer Anfrage eine Nachricht zu generieren, die an den Server geschickt werden kann. Der Aufruf eines Unterprogramms im **Client-Stub** löst eine so genannte Ausnahmebehandlung aus: zunächst werden die Parameter des Aufrufs in eine Nachricht verpackt. Die `send`-Primitive beauftragt dann den Kern, die Nachricht an den Server zu schicken (siehe Abbildung 12.6). Im Anschluss daran ruft der Client-Stub ein `receive` auf und blockiert so lange, bis eine Antwort eintrifft.

Nun zur Server-Seite: trifft die Nachricht vom Client ein, so übergibt der Kern diese an den so genannten **Server-Stub**, der an den aktuellen Server gebunden ist. Im Allgemeinen hat der Server-Stub gerade ein `receive` aufgerufen und wartet auf ankommende Nachrichten. Der Server-Stub packt dann die in der Nachricht enthaltenen Parameter aus und ruft das Unterprogramm lokal auf. Parameter und Rücksprungadresse werden abgelegt. Es folgt die Ausführung und die Rückgabe der Ergebnisse an den Server-Stub. Erhält der Server-Stub die Kontrolle zurück, so verpackt er die Ergebnisse wieder zu einer Nachricht, die er mittels `send` an den Client schickt (siehe Abbildung 12.7). Gegebenenfalls ruft er danach

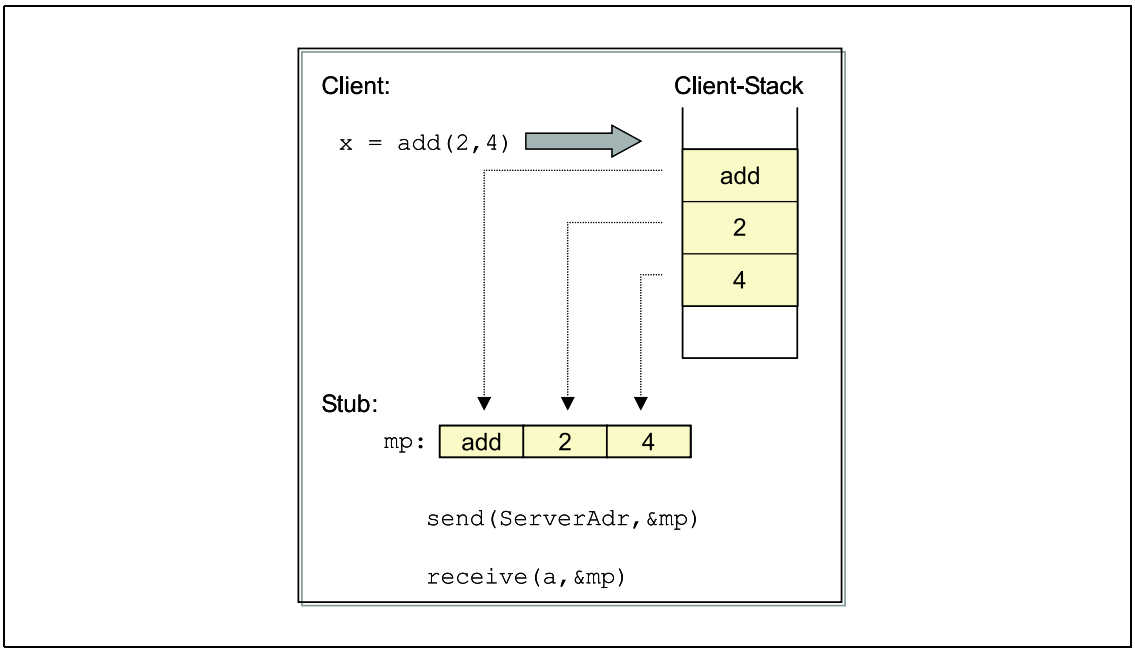


Abbildung 12.6: Funktionsweise von RPC: Schritt 1

wieder ein `receive` auf, um für den Empfang der nächsten Nachricht bereit zu sein.

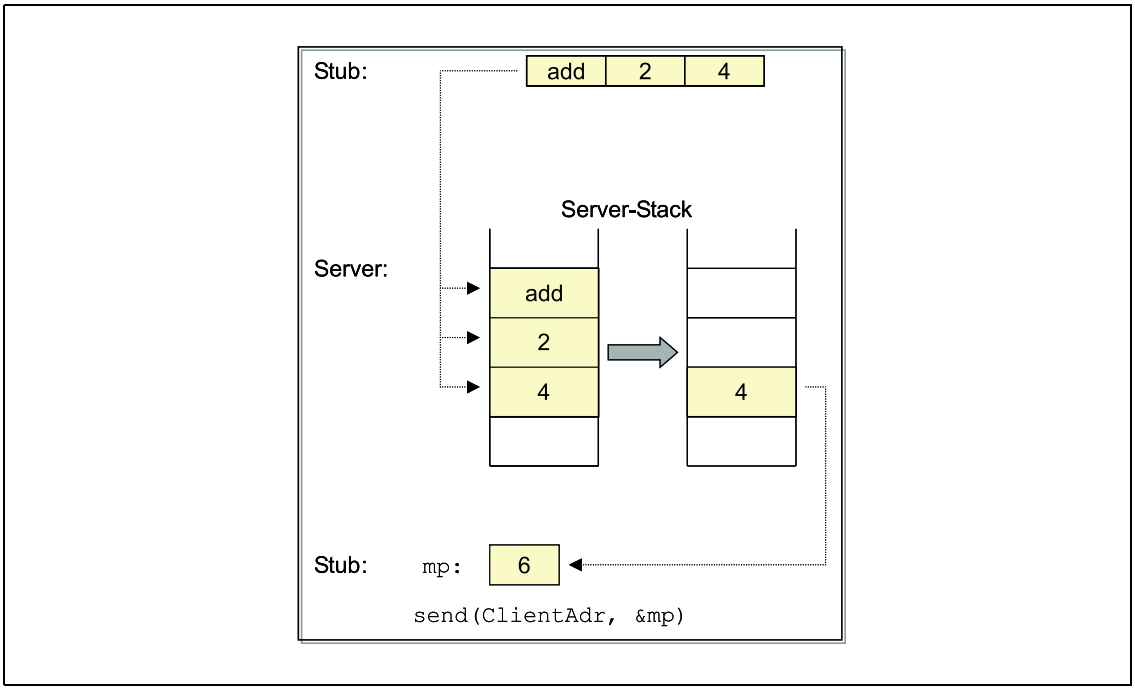


Abbildung 12.7: Funktionsweise von RPC: Schritt 2

Erreicht die Antwort des Servers den Client, wird sie in einen entsprechenden Puffer kopiert und der Client-Stub wird entblockiert. Der Client-Stub prüft die Nachricht und packt sie aus. Dann schiebt er sie auf den Stack (siehe Abbildung 12.8). Wenn der Aufrufer von `read` die Kontrolle zurückerhält, so ist ihm nur bekannt, dass die Daten vorliegen, nicht aber, dass ein Aufruf entfernt ausgeführt wurde. Diese Transparenz zeichnet den RPC aus.

Auf diese Art und Weise können entfernte Dienste durch einfache, d.h. lokale Unterpro-

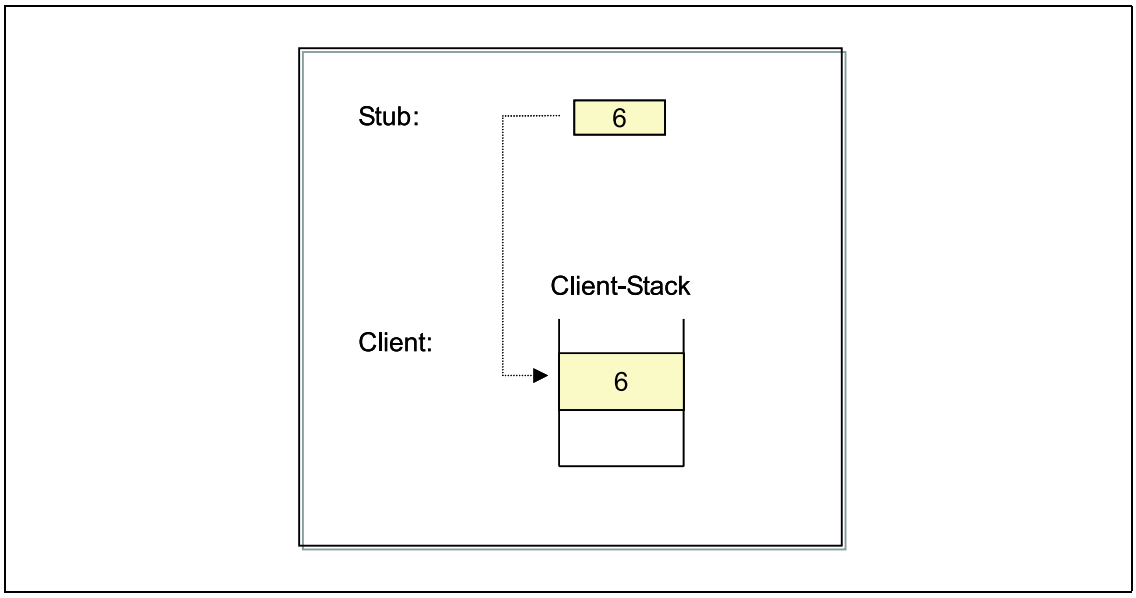


Abbildung 12.8: Funktionsweise von RPC: Schritt 3

grammaufrufe ausgeführt werden. Dies erfolgt ohne explizite Anwendung der Kommunikationsprimitive. Alle Details sind durch zwei Bibliotheksroutinen – den Client- und den Server-Stub – verborgen. Eine Zusammenfassung dieser Schritte ist in Abbildung 12.9 dargestellt.

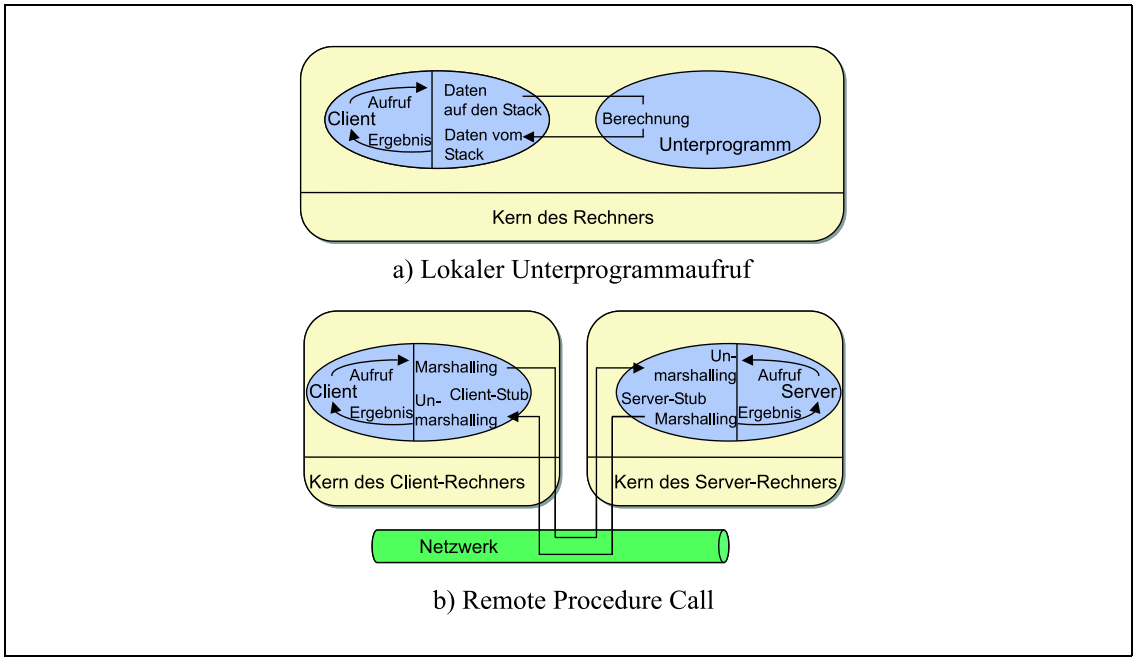


Abbildung 12.9: Das Prinzip des RPCs im Vergleich zum lokalen Unterprogrammaufruf

Es gibt verschiedene Probleme, die bei der Realisierung eines RPCs zu lösen sind. Eines der wichtigsten ist das so genannte **Parameter Marshalling**, die Parameterübergabe. Hierunter versteht man das Verpacken einer Nachricht. Zum Beispiel würde aus dem einfachen Aufruf `x=produkt(2,y)` eine Nachricht der Form `(produkt,2,y)` oder `(y,2,produkt)` gemacht werden. Was in diesem Falle noch recht unkritisch aussieht, kann sich schnell als Problem herausstellen: unterschiedliche Formate (INTEL-Format, SPARC-

Format) können unterschiedliche Interpretationen der Nachricht bedingen.

Ein weiteres Problem, das bei Verteilungsplattformen gelöst sein muss, ist die Thematik des **dynamischen Bindens**: In einem Verteilten System ist es möglich, dass sich aufgrund bestimmter Ereignisse die Adresse eines Servers bzw. einer Schnittstelle ändert. Dies hätte zur Folge, dass zahlreiche Programme, die mit Konstanten als Adresse arbeiten, neu geschrieben und übersetzt werden müssten. Dieses Problem wird durch Einsatz so genannter *dynamischer Binder* gelöst.

Zu Beginn einer Server-Ausführung erfolgt dabei der Aufruf von `initialize()`. Dies bewirkt das Exportieren der Schnittstelle des Servers. Der Server sendet hierzu eine Nachricht an ein Programm, den Binder und gibt damit seine Existenz bekannt. Dieser Vorgang wird auch als Registrierung des Servers bezeichnet. Dabei übergibt der Server dem Binder seinen Namen, die Versionsnummer (bzw. einen eindeutigen Bezeichner) und ein so genanntes Handle, das zur Lokalisierung des Servers dient, z.B. eine Ethernet- oder IP-Adresse. Möchte der Server irgendwann keinen Dienst mehr anbieten, so lässt er sich einfach deregistrieren.

Mit diesen Voraussetzungen auf Server-Seite ist dynamisches Binden möglich: Ruft der Client zum ersten Mal ein entferntes Unterprogramm auf, so erkennt der zugehörige Stub, dass der Client noch nicht an einen entsprechenden Server gebunden ist. Dann sendet der Client-Stub eine Nachricht an den Binder, um z.B. eine bestimmte Version einer bestimmten Schnittstelle zu importieren. Der Binder überprüft, ob ein oder mehrere Server eine Schnittstelle mit diesem Namen und der entsprechenden Versionsnummer exportieren. Ist das nicht der Fall, so schlägt die Operation fehl. Existiert jedoch ein passender Server, so liefert der Binder das Handle und den eindeutigen Bezeichner an den Client-Stub zurück. Der Client-Stub benutzt das Handle als Adresse, an die er die Anfragenachricht sendet. Die Nachricht enthält die Parameter und den eindeutigen Bezeichner, mit dem der richtige Server ausgewählt wird, falls mehrere existieren.

Der Vorteil des dynamischen Bindens liegt in der hohen Flexibilität. Der Binder kann seine Server in regelmäßigen Abständen überprüfen und ggf. deregistrieren, Lastausgleiche steuern, wenn mehrere Server den gleichen Dienst anbieten (**Load Balancing**) und Sicherheitsmechanismen integrieren (z.B. Authentifikation). Der Preis für diese zusätzliche Funktionalität ist der Aufwand für das Im- und Exportieren von Schnittstellen. Außerdem kann der Binder in großen Systemen auch zu einem Engpass werden, sodass mehrere Binder benötigt werden, die durch ihr zu synchronisierendes Verhalten eine hohe Netzlast bedingen können.

Index

N

O

P

R

S

T

U

V

W

Z